**OOPs** (Object-Oriented Programming System)

**Object** means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts: **Inheritance, Polymorphism, Abstraction, Encapsulation**

Apart from these concepts, there are some other terms which are used in Object-Oriented design: **Coupling, Cohesion, Association, Aggregation, Composition**  
  
**Object**: Any entity that has state and behavior is known as an object. An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory.

**Class**: Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.  
  
**Inheritance**: When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.  
  
**Rules of Inheritance in Java**

RULE 1: Multiple Inheritance is NOT permitted in Java.

RULE 2: Cyclic Inheritance is NOT permitted in Java.

RULE 3: Private members do NOT get inherited.

RULE 4: Constructors cannot be inherited in Java.

RULE 5: In Java, we assign parent reference to child objects.

**Polymorphism**: If one task is performed in different ways, it is known as polymorphism. In Java, we use method overloading and method overriding to achieve polymorphism.  
  
**Abstraction**: Hiding internal details and showing functionality is known as abstraction. In Java, we use abstract class and interface to achieve abstraction.  
  
**Encapsulation**: Binding (or wrapping) code and data together into a single unit are known as encapsulation. A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.  
  
**Coupling**: Coupling refers to the knowledge or information or dependency of another class. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can **use interfaces for the weaker coupling** because there is no concrete implementation.

**Cohesion**: ***Cohesion refers to the level of a component which performs a single well-defined task***. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.  
  
**Cohesion** is a fundamental principle of software design that measures ***how closely related and focused the responsibilities of a module, class, or method are***.  
**High cohesion** means a class or method has a well-defined, single responsibility and is focused on a specific task.  
**Low cohesion** means a class or method is doing multiple, unrelated tasks, making the code harder to maintain, understand, and reuse. *Cohesion is directly related to the Single Responsibility Principle (SRP) in SOLID principles.*  
 **Cohesion** is an indication of ***how related and focused the responsibilities of a software element are***.  
**Coupling** refers to how strongly a software element is connected to other elements.  
The software element could be class, package, component, subsystem or a system. And while designing the systems it is recommended to have software elements that have **High cohesion** and **Low coupling**.  
  
**Association**: Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects: ***One to One, One to Many, Many to One, Many to Many*.** Association can be unidirectional or bidirectional.

**Aggregation**: Aggregation is **a way to achieve Association**. Aggregation represents the relationship where one object contains other objects as a part of its state. ***It represents the weak relationship between objects***. It is also termed as a **has-a** relationship in Java. Like, inheritance represents the is-a relationship. It is another way to reuse objects.

**Composition**: The composition is also a **way to achieve Association**. The composition represents the relationship where one object contains other objects as a part of its state. ***There is a strong relationship between the containing object and the dependent object***. It is the state where containing objects do not have an independent existence. If you delete the parent object, all the child objects will be deleted automatically.  
   
**General contract associated with hashCode() method**

The**hashCode()** method should return the same integer value for the same object for each calling of this method unless the value stored in the object is modified.

If two objects are equal (according to **equals()** method) then the **hashCode()** method should return the same integer value for both the objects.

But, it is not necessary that the**hashCode()** method will return the distinct result for the objects that are not equal (according to **equals()** method.

1. **What if : Overriding only equals() without overriding hashCode():**

* Causes the two equal instances to have unequal hash codes that is in violation of the hashCode contract which we said above.
* Since the default hashCode implementation in the Object class return distinct integers for distinct objects, if only equals() method is overridden, emp1 will be placed in one bucket, emp2 will be placed in another bucket. So emp1.hashCode() != emp2.hashCode(). ***So even though both emp1 and emp2 are equal, they don’t hash to the same bucket and both of them reside in the collection as separate keys***.
* It is worth noting that if the class instance is never used in any hash-based collections, then it doesn’t really matter if hashCode() is overridden or not.

**2. What if: Overriding only hashCode() without overriding equals():**

* Then, emp1 and emp2 objects will hash to the same bucket as they produces the same hash code. But since equals() is not overridden, when emp2 is added to Set, it iterates through bucket looking if there is Employee such as emp2 i.e.: emp2.equals(emp1) will be false.
* If we only override the hashCode() method, both e1 and e2 will hash to the same bucket as they produce the same hash code. But since the equals() method is not overridden, when the set hashes e2 and iterates through the bucket looking if there is an Employee e such that e2.equals(e1) is true, it won’t find any as e2.equals(e1) will be false.
* If o1.equals(o2), then o1.hashCode() == o2.hashCode() should always be true.
* If o1.hashCode() == o2.hashCode is true, it doesn’t mean that o1.equals(o2) will be true.

Please note that even though equal objects must have equal hash codes, the reverse is not true. It is perfectly valid to override hashCode() without overriding equals() as objects with equal hash codes need not be equal.  
  
***What is Method Hiding in Java?***  
**Definition**: Method Hiding occurs when a static method in a parent class is redefined in a child class using the same method signature. Instead of overriding, the child class hides the parent class method.

**Key Points:**

1️ only static methods can be hidden, instance methods are always overridden.

2 Binding happens at compile time (early binding) instead of runtime (late binding).

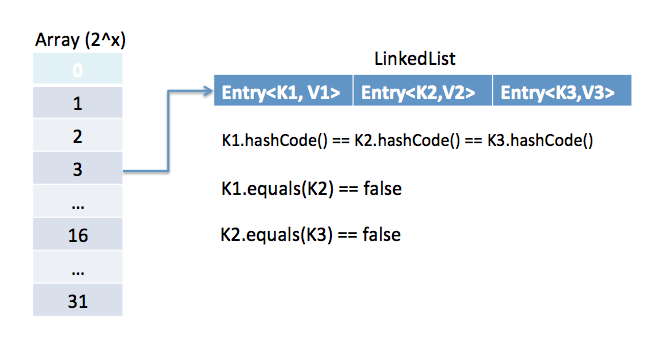
3 Method invocation depends on the reference type, not the object type.  
  
Eg: Parent obj3 = new Child(); // Reference of Parent, object of Child

obj3.show(); // Calls Parent's show() method (Method hiding, NOT overridden)

[**What is Hash Collision**](https://www.digitalocean.com/community/tutorials/java-equals-hashcode#what-is-hash-collision)

In very simple terms, Java Hash table implementations uses following logic for get and put operations.

* First identify the “Bucket” to use using the “key” hash code.
* If there are no objects present in the bucket with same hash code, then add the object for put operation and return null for get operation.
* If there are other objects in the bucket with same hash code, then “key” equals method comes into play.
  + If equals() return **true** and it’s a **put** operation, then object value is **overridden**.
  + If equals() return **false** and it’s a **put** operation, then **new entry is added** to the bucket.
  + If equals() return **true** and it’s a **get** operation, then object **value** is returned.
  + If equals() return **false** and it’s a **get** operation, then **null** is returned.

Below image shows a bucket items of HashMap and how their equals() and hashCode() are related.  
The phenomenon **when two keys have same hash code is called hash collision**. If hashCode() method is not implemented properly, there will be higher number of hash collision and map entries will not be properly distributed causing slowness in the get and put operations. This is the reason for prime number usage in generating hash code so that map entries are properly distributed across all the buckets.

**Fail-fast and Fail-safe iterations in Java Collections** Using iterations we can traverse over the collections objects. The iterators can be either fail-safe or fail-fast.

**Fail-safe** iterator’s means they **will not throw any exception** even if the collection is modified while iterating over it.  
However if we use an fail-safe collection e.g. **CopyOnWriteArrayList** then no exception will occur **Fail-fast** iterators **throw an exception** (ConcurrentModificationException) if the collection is modified while iterating over it.

***Fail-Fast Iterators internal working***  
 every fail fast collection has a **modCount**field, to represent how many times the collection has changed/modified.  
So at every modification of this collection we increment the modCount value. For example the modCount is incremented in below cases:  
1. When one or more elements are removed.  
2. When one or more elements are added.  
3. When the collection is replaced with other collection.  
4. When the collection is sorted.  
So every time there is some change in the collection structure, the mod count is incremented.  
***Fail-Safe Iterators internal working***

Unlike the fail-fast iterators, these iterators ***traverse over the clone of the collection***. So even if the original collection gets structurally modified, no exception will be thrown.

E.g. in case of CopyOnWriteArrayList the original collections is passed and is stored in the iterator  
  
<https://anmolsehgal.medium.com/fail-fast-and-fail-safe-iterations-in-java-collections-11ce8ca4180e>

**ConcurrentHashMap**Uses an **array of Node<K, V>** (similar to HashMap)

Uses **CAS (Compare-And-Swap)** and **fine-grained synchronization**

**No segment-based locking** (like in Java 7)

Supports **lock-free reads** and **bucket-level locking** for writes

**Locking Mechanism – Step by Step**

**✅ 1. Reading (get())**

**Lock-free**

Just reads the table at index (n - 1) & hash

Uses **volatile read** (ensures visibility)

map.get("key"); // completely lock-free

**✅ 2. Writing (put())**

**a. If bucket is empty:**

Use **CAS** to insert the node

if (CAS(bucket, null, new Node(k, v)))

return; // success without locking

**b. If bucket is non-empty:**

Use synchronized **only on that bucket’s first node**

Other threads can still modify other buckets concurrently

synchronized (firstNode) {

// do insert/update inside the bucket

}

**c. If entries in bucket ≥ 8, it is treeified (turned into a balanced tree), and locking is done on the TreeBin.**

ConcurrentHashMap uses **bucket-level locking** and **CAS** for writes

**Reads are lock-free**

Only **locks the affected bucket** using synchronized

In Java 8+, it's **more scalable** than segmented approach (Java 7)

Resize is done **concurrently** by multiple threads

**What is Java Architecture?**

* In Java, there is a process of compilation and interpretation.
* The code written in [Java](https://www.edureka.co/blog/java-tutorial/), is converted into byte codes which is done by the Java Compiler.
* The byte codes, then are converted into machine code by the JVM.
* The Machine code is executed directly by the machine.

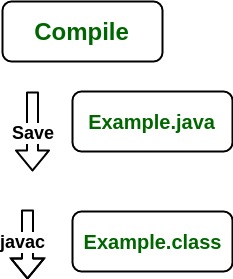
This diagram illustrates the internal working of a Java code, or precisely, Java Architecture!

|  |  |
| --- | --- |
| JVM - Java Architecture - Edureka | difference **Components of Java Architecture** There are three main components of Java language: JVM, JRE, and JDK. |

* **JDK** – **Java Development Kit** which provides the environment to **develop and execute (run)** the Java program. JDK is a kit (or package) which includes two things
  + 1. Development Tools (to provide an environment to develop your java programs)
    2. JRE (to execute your java program).

**Note:** JDK is only used by Java Developers.

* **JRE** – **Java Runtime Environment** is an installation package which provides environment to **only run (not develop)** the java program (or application) onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
* **JVM** – **Java Virtual machine** is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for **executing the java program line by line** hence it is also known as interpreter.

**How does JRE works?**  
To understand how the JRE works let us consider a Java source file saved as *Example.java*. The file is compiled into a set of Byte Code that is stored in a “*.class*” file. Here it will be “*Example.class*“.  
  
  
  
The following diagram depicts what is done at compile time.  
  
The following actions occur at runtime.

* **Class Loader** loads all necessary classes needed for the execution of a program. It provides security by separating the namespaces of the local file system from that imported through the network. These files are loaded either from a hard disk, a network or from other sources.
* **Byte Code Verifier** the JVM puts the code through the Byte Code Verifier that checks the format and checks for an illegal code. Illegal code, for example, is code that violates access rights on objects or violates the implementation of pointers.  
  The Byte Code verifier ensures that the code adheres to the JVM specification and does not violate system integrity.
* **Interpreter** at runtime the Byte Code is loaded, checked and run by the interpreter. The interpreter has the following two functions:
  + Execute the Byte Code
  + Make appropriate calls to the underlying hardware  
      
    Both operations can be shown as:

|  |
| --- |
| interpreter and runtime interaction |

To understand the interactions between JDK and JRE consider the following diagram.  
 ***How JVM Works – JVM Architecture?***JVM acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE.  
  
When we compile a .java file, .class files (contains byte-code) with the same class names present in .java file are generated by the Java compiler. This .class file goes into various steps when we run it. These steps together describe the whole JVM.

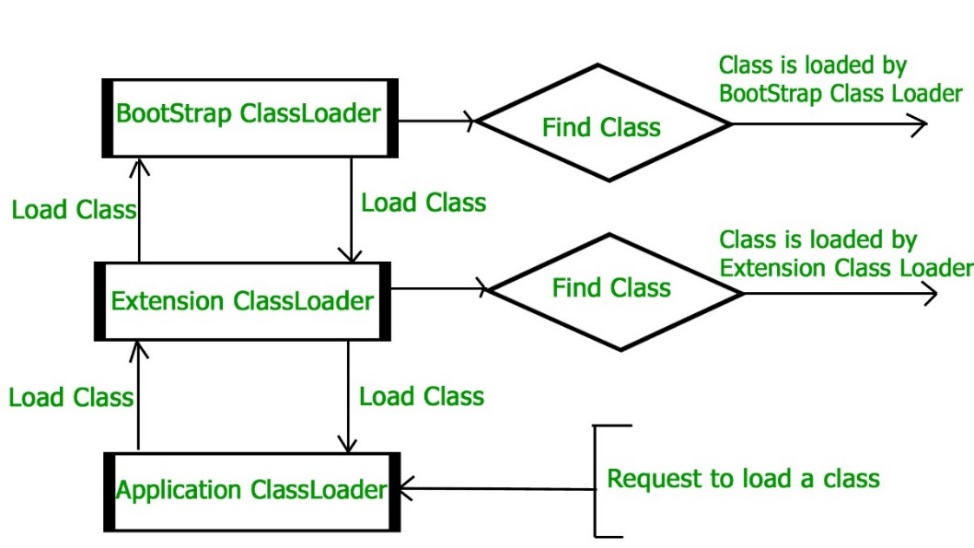
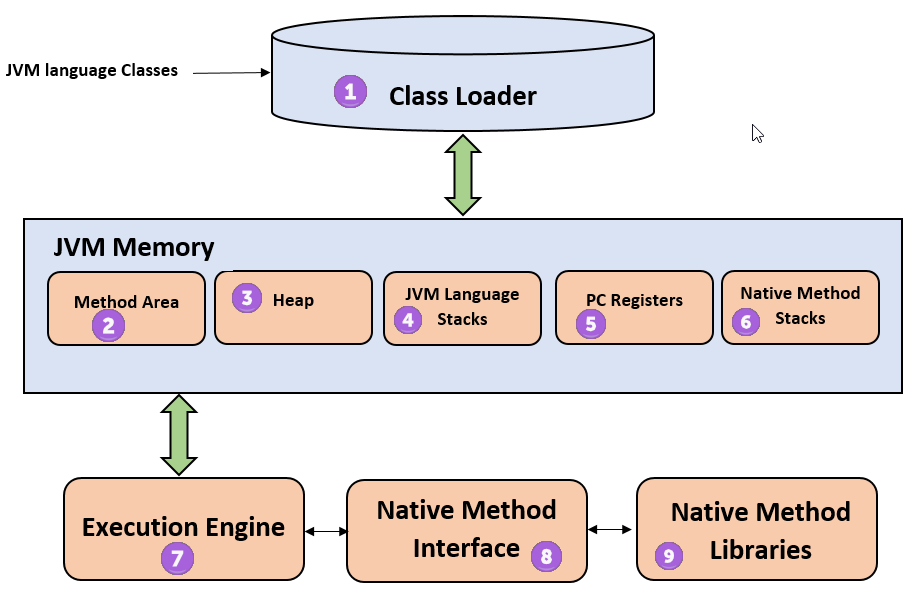
|  |  |
| --- | --- |
| jvm | JVM - Java Architecture - Edureka |

**Class Loader Subsystem** It is mainly responsible for three activities. ***Loading, Linking, Initialization***  
  
**Loading**: The Class loader reads the “.class” file, generate the corresponding binary data and save it in the method area. For each “.class” file.  
  
**Linking**: Performs verification, preparation, and (optionally) resolution.

1. **Verification**: It ensures the correctness of the .class file i.e. it checks whether this file is properly formatted and generated by a valid compiler or not. If verification fails, we get run-time exception java.lang.VerifyError. This activity is done by the component ByteCodeVerifier. Once this activity is completed then the class file is ready for compilation.
2. **Preparation**: JVM allocates memory for class variables and initializing the memory to default values.
3. **Resolution**: It is the process of replacing symbolic references from the type with direct references. It is done by searching into the method area to locate the referenced entity.

**Initialization**: In this phase, all static variables are assigned with their values defined in the code and static block (if any). This is executed from top to bottom in a class and from parent to child in the class hierarchy.   
  
There are three class loaders: ***Bootstrap, Extension and System***

1. **Bootstrap** class loader: Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. ***It loads core java API classes present in the “JAVA\_HOME/jre/lib” directory***. This path is popularly known as the bootstrap path. It is implemented in native languages like C, C++.
2. **Extension** class loader: It is a child of the bootstrap class loader. ***It loads the classes present in the extensions directories “JAVA\_HOME/jre/lib/ext”***(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the sun.misc.Launcher$ExtClassLoader class.
3. **System/Application** class loader: It is a child of the extension class loader. It is responsible to ***load classes from the application classpath***. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the sun.misc.Launcher$AppClassLoader class.

**Note**: ***JVM follows the Delegation-Hierarchy principle to load classes***. System class loader delegate load request to extension class loader and extension class loader delegate request to the bootstrap class loader. If a class found in the boot-strap path, the class is loaded otherwise request again transfers to the extension class loader and then to the system class loader. At last, if the system class loader fails to load class, then we get run-time exception java.lang.ClassNotFoundException.  
  
ClassNotFoundException and NoClassDefFoundError occur when a particular class is not found at runtime. However, they occur at different scenarios.  
  
**ClassNotFoundException** is an exception that occurs when you try to load a class at run time using Class.forName() or loadClass() methods and mentioned classes are not found in the classpath.  
**NoClassDefFoundError** is an error that occurs when a particular class is present at compile time, but was missing at run time.  
 **1) ClassLoader** The class loader is a subsystem used for loading class files. It performs three major functions viz. ***Loading, Linking, and Initialization***.

**2) Method Area** JVM Method Area ***stores class structures like metadata, the constant runtime pool, and the code for methods***.

**3) Heap** All the Objects, their ***related instance variables, and arrays are stored in the heap***. This memory is common and shared across multiple threads.

**4) JVM language Stacks** Java language Stacks store local variables, and its partial results. Each thread has its own JVM stack, created simultaneously as the thread is created. A new frame is created whenever a method is invoked, and it is deleted when method invocation process is complete.

**Stack Area**: Stack Area generates when a thread creates. It can be of either fixed or dynamic size. The stack memory is ***allocated per thread***. It is used to ***store data and partial results***. It contains references to heap objects. It also holds the value itself rather than a reference to an object from the heap. The variables which are stored in the stack have certain visibility, called scope.

**Stack Frame:** Stack frame is a data structure that contains the thread’s data. Thread data represents the state of the thread in the current method.

* It is used to store partial results and data. It also performs dynamic linking, values return by methods and dispatch exceptions.
* When a method invokes, a new frame creates. It destroys the frame when the invocation of the method completes.
* Each frame contains own Local Variable Array (LVA), Operand Stack (OS), and Frame Data (FD).
* The sizes of LVA, OS, and FD determined at compile time.
* Only one frame (the frame for executing method) is active at any point in a given thread of control. This frame is called the current frame, and its method is known as the current method. The class of method is called the current class.
* The frame stops the current method, if its method invokes another method or if the method completes.
* The frame created by a thread is local to that thread and cannot be referenced by any other thread.

**5)  PC Registers** PC register store the address of the Java virtual machine instruction which is currently executing. In Java, each thread has its separate PC register.

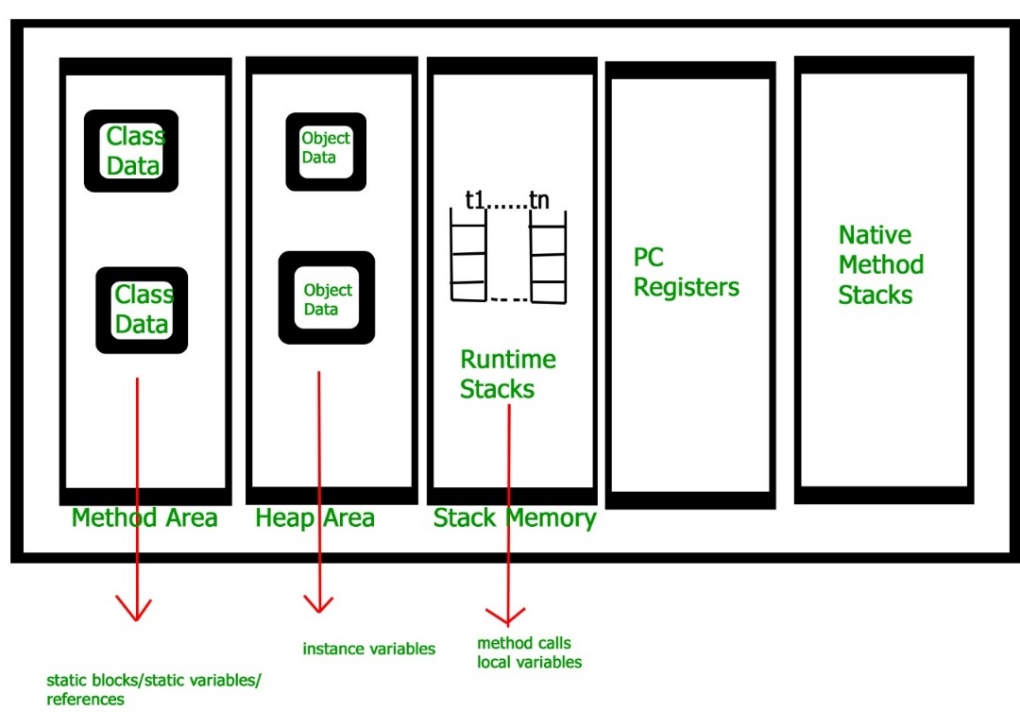
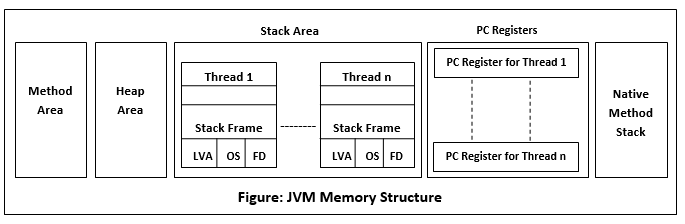
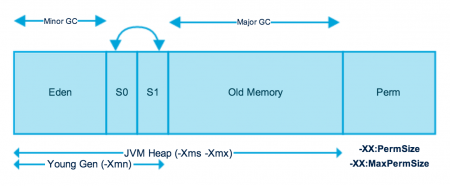
**6) Native Method Stacks** Native method stacks hold the instruction of native code depends on the native library. It is written in another language instead of Java.

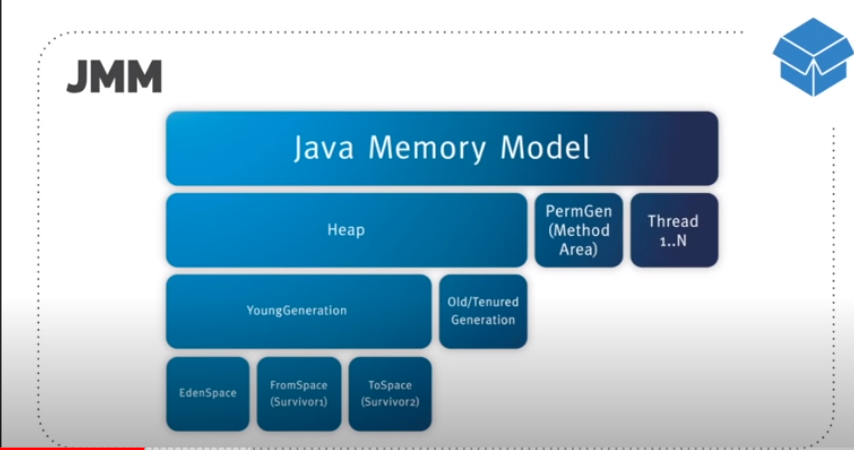
**7) Execution Engine** It is a type of software used to test hardware, software, or complete systems. The test execution engine never carries any information about the tested product.

**8) Native Method interface** The Native Method Interface is a programming framework. It allows Java code which is running in a JVM to call by libraries and native applications.

**9) Native Method Libraries** Native Libraries is a collection of the Native Libraries(C, C++) which are needed by the Execution Engine. **JVM Memory Java (JVM) Memory Model**

1. ***Method area***: In the method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is **only one method area per JVM**, and it is a shared resource.
2. **Heap area**: Information of all objects is stored in the heap area. There is also **one Heap Area per JVM**. It is also a shared resource.
3. **Stack area**: For **every thread, JVM creates one run-time stack** which is stored here. Every block of this stack is called activation record/stack frame which stores methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminates, its run-time stack will be destroyed by JVM. It is not a shared resource.
4. **PC Registers**: Store address of current execution instruction of a thread. Obviously, **each thread has separate PC Registers**.
5. **Native method stacks**: For **every thread, a separate native stack is created**. It stores native method information.

  
  
  
  
  
<https://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java>



**Execution Engine** executes the “.class” (bytecode). It reads the byte-code line by line, uses data and information present in various memory area and executes instructions. It can be classified into three parts:  
 **Interpreter**: It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.  
 **Just-In-Time Compiler (JIT)**: It is used to increase the efficiency of an interpreter. It compiles the entire bytecode and changes it to native code so whenever the interpreter sees repeated method calls, JIT provides direct native code for that part so re-interpretation is not required, thus efficiency is improved.  
 **Garbage Collector**: It destroys un-referenced objects.

**Software Code Compilation & Execution process** In order to write and execute a software program, you need the following  
**1) Editor**– To type your program into, a notepad could be used for this  
**2) Compiler**– To convert your high language program into native machine code  
**3) Linker**– To combine different program files reference in your main program together.  
**4) Loader**– To load the files from your secondary storage device like Hard Disk, Flash Drive, CD into RAM for execution. The loading is automatically done when you execute your code.  
**5) Execution** – Actual execution of the code which is handled by your OS & processor  
  
**what does Java Garbage Collector?**

JVM controls the garbage collector. JVM decides when to perform the garbage collection. We can also request to the JVM to run the garbage collector. But there is no guarantee under any conditions that the JVM will comply. JVM runs the garbage collector if it senses that memory is running low. When Java program request for the garbage collector, the JVM usually grants the request in short order. It does not make sure that the requests accept.

The point to understand is that "**when an object becomes eligible for garbage collection?**"

Every Java program has more than one thread. Each thread has its execution stack. There is a thread to run in Java program that is a main() method. Now we can say that an object is eligible for garbage collection when no live thread can access it. The garbage collector considers that object as eligible for deletion. If a program has a reference variable that refers to an object, that reference variable available to live thread, this object is called **reachable**.

Here a question arises that "**Can a Java application run out of memory?**"  
The answer is yes. The garbage collection system attempts to objects from the memory when they are not in use. Though, if you are maintaining many live objects, garbage collection does not guarantee that there is enough memory. Only available memory will be managed effectively.

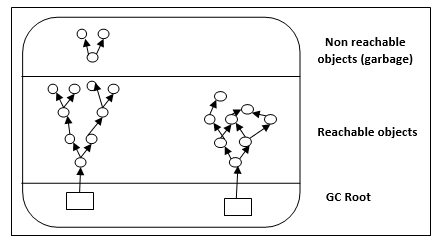
### **Types of Garbage Collection** There are five types of garbage collection are as follows:

* **Serial GC:** It uses the mark and sweeps approach for young and old generations, which is minor and major GC.
* **Parallel GC:** It is similar to serial GC except that, it spawns N (the number of CPU cores in the system) threads for young generation garbage collection.
* **Parallel Old GC:** It is similar to parallel GC, except that it uses multiple threads for both generations.
* **Concurrent Mark Sweep (CMS) Collector:** It does the garbage collection for the old generation. You can limit the number of threads in CMS collector using **XX:ParalleCMSThreads=JVM option**. It is also known as Concurrent Low Pause Collector.
* **G1 Garbage Collector:** It introduced in Java 7. Its objective is to replace the CMS collector. It is a parallel, concurrent, and CMS collector. There is no young and old generation space. It divides the heap into several equal sized heaps. It first collects the regions with lesser live data.

## Mark and Sweep Algorithm

JRockit JVM uses the mark, and sweep algorithm for performing the garbage collection. It contains two phases, the mark phase, and the sweep phase.

**Mark Phase:** Objects that are accessible from the threads, native handles, and other GC root sources are marked as live. Every object tree has more than one root objects. GC root is always reachable. So any object that has a garbage collection root at its root. It identifies and marks all objects that are in use, and the remaining can be considered garbage.



**Sweep Phase:** In this phase, the heap is traversed to find the gap between the live objects. These gaps are recorded in the free list and are available for new object allocation.

There are two improved versions of mark and sweep:

* **Concurrent Mark and Sweep**
* **Parallel Mark and Sweep**

### **Concurrent Mark and Sweep**

It allows the threads to continue running during a large portion of the garbage collection. There are following types of marking:

* **Initial marking:** It identifies the root set of live objects. It is done while threads are paused.
* **Concurrent marking:** In this marking, the reference from the root set are followed. It finds and marks the rest of the live objects in a heap. It is done while the thread is running.
* **Pre-cleaning marking:** It identifies the changes made by concurrent marking. Other live objects marked and found. It is done while the threads are running.
* **Final marking:** It identifies the changes made by pre-cleaning marking. Other live objects marked and found. It is done while threads are paused.

**Parallel Mark and Sweep**: It uses all available CPU in the system for performing the garbage collection as fast as possible. It is also called the parallel garbage collector. Threads do not execute when the parallel garbage collection executes.

**Pros of Mark and Sweep**

* It is a recurring process.
* It is an infinite loop.
* No additional overheads allowed during the execution of an algorithm.

**Cons of Mark and Sweep**

* It stops the normal program execution while the garbage collection algorithm runs.

It runs multiple times on a program.

***Garbage Collector Evolution by Java Version***

|  |  |  |  |
| --- | --- | --- | --- |
| **Java Version** | **Default GC** | **New/Improved GCs** | **Notes** |
| **Java 8** | **Parallel GC** (Throughput Collector) | G1 GC (optional) | G1 was introduced but not default. |
| **Java 11** | **G1 GC** | ✅ ZGC (Experimental) | Focus on low-pause times; ZGC for ultra-low latency. |
|  |  | ✅ Epsilon GC |  |
| **Java 17** | **G1 GC** | ✅ ZGC (Stable) | More options for ultra-low pause collectors. |
|  |  | ✅ Shenandoah (Experimental) |  |
|  |  | ✅ Epsilon |  |
| **Java 21** | **G1 GC** (Still default) | ✅ ZGC, ✅ Shenandoah (Stable) | Focus on throughput + low latency. |
|  |  | ✅ Generational ZGC (Preview) | JEP 439 for Gen ZGC. |

<https://www.studytonight.com/java/string-handling-in-java.php> **String** is an **object** that represents sequence of characters. In Java, String is represented by String class which is located into java.lang package

It is probably the most commonly used class in java library. In java, every string that we create is actually an object of type **String**. One important thing to notice about string object is that *string objects are****immutable****that means once a string object is created it cannot be changed*.

The Java String class implements Serializable, Comparable and CharSequence interface that we have represented using the below image.

|  |  |
| --- | --- |
| string-handling-feature | char-string-handling-feature |
|  |  |

In Java, **CharSequence** Interface is used for representing a sequence of characters. CharSequence interface is implemented by String, StringBuffer and StringBuilder classes. This three classes can be used for creating strings in java.

**What is an Immutable object**: *An object whose state cannot be changed after it is created is known as an Immutable object*. String, Integer, Byte, Short, Float, Double and all other wrapper classes objects are immutable.

<https://www.studytonight.com/java/multithreading-in-java.php>  
  
***Java String Creation and Memory Internals***

String s1 = new String ("RadhaKrishna"); String s2 = "RadhaKrishna";

**What happens internally in java?**

In Java, there's an important difference between how these two String objects are created and stored in memory. Let me explain what happens internally:

1. String s1 = **new String ("RadhaKrishna")**;

- This **creates a new String object in the heap memory**

- First, the ***literal "RadhaKrishna" is placed in the string constant pool (if not already present)***

- Then a ***new String object is created in the heap that refers to this literal***

- *This means you* ***always get a new object****,* ***even if an identical string exists in the pool***

2. String s2 = **"RadhaKrishna";**

- This uses string literal syntax and checks the string constant pool first

- If "RadhaKrishna" already exists in the pool, s2 will reference that existing object

- If not, a new String object is created in the pool

- This is more memory efficient as it can reuse existing strings

Key differences:

- Memory location: s1 is in heap, s2 may be in the string pool (which is part of heap but specially managed)

- Object creation: s1 always creates new object, s2 may reuse existing one

- Performance: s2 is generally more efficient as it can avoid duplicate strings

Example showing the difference:

String s1 = new String ("RadhaKrishna");

String s2 = "RadhaKrishna";

String s3 = "RadhaKrishna";

System.out.println(s1 == s2); // false - different objects

System.out.println(s2 == s3); // true - same object from pool

Best practice:

- Use string literal syntax (String s = "...") when possible for better performance

- Only use new String() when you explicitly need a new object instance

***Understanding intern() Method in Java***

The intern() method in Java helps reduce memory usage by ensuring that identical string literals share the same memory reference.

***What Does intern() Do?***

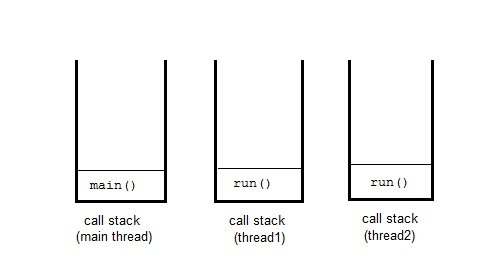
• intern() moves a string to the String Constant Pool (SCP).

• If the SCP already contains the same string, it returns a reference to that existing string.

• If the SCP does not contain the string, it adds it to the pool and returns the reference.  
  
**Multithreading in Java:** Multithreading is a concept of running multiple threads simultaneously. Thread is a lightweight unit of a process that executes in multithreading environment.  
  
Multithreaded programs contain two or more threads that can run concurrently and each thread defines a separate path of execution. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.  
  
In Java, the word **thread** means two different things.

* An instance of **Thread** class.
* or, A thread of execution.

An instance of **Thread** class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.  
  
**Concurrency** is when two or more tasks can start, run, and complete in overlapping time **periods**. It doesn't necessarily mean they'll ever both be running **at the same instant**. For example, *multitasking on a single-core machine*.

**Parallelism** is when tasks *literally* run at the same time, e.g., on a *multicore processor*.  
  
  
**Advantage of Multithreading**Multithreading **reduces** the CPU **idle time** that increase overall performance of the system. Since thread is lightweight process then it takes **less memory** and perform **context switching** as well that helps to share the memory and reduce time of switching between threads.  
  
**Multitasking**Multitasking is a process of performing multiple tasks simultaneously. We can understand it by computer system that perform multiple tasks like: writing data to a file, playing music, downloading file from remote server at the same time.  
  
Multitasking can be achieved either by using multiprocessing or multithreading. Multitasking by using multiprocessing involves multiple processes to execute multiple tasks simultaneously whereas Multithreading involves multiple threads to executes multiple tasks.  
  
**Why Multithreading?**  
Thread has many advantages over the process to perform multitasking. Process is heavy weight, takes more memory and occupy CPU for longer time that may lead to performance issue with the system. To overcome these issue process is broken into small unit of independent sub-process. These sub-process are called threads that can perform independent task efficiently. So nowadays computer systems prefer to use thread over the process and use multithreading to perform multitasking.  
 **How to Create Thread ?**To create a thread, Java provides a class **Thread** and an interface **Runnable** both are located into java.lang package.  
  
*We can create thread either by extending Thread class or implementing Runnable interface*. Both includes a run method that must be override to provide thread implementation.  
  
It is recommended to use Runnable interface if you just want to create a thread but can use Thread class for implementation of other thread functionalities as well.  
 **Life cycle of a Thread**Like process, thread have its life cycle that includes various phases like: new, running, terminated etc. we have described it using the below image.



1. **New:** A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. **Runnable:** After invocation of start() method on new thread, the thread becomes runnable.
3. **Running:** A thread is in running state if the thread scheduler has selected it.
4. **Waiting:** A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.
5. **Terminated :** A thread enter the terminated state when it complete its task.

**Daemon Thread**Daemon threads is a low priority thread that provide supports to user threads. These threads can be user defined and system defined as well. ***Garbage collection thread is one of the system generated daemon thread that runs in background***. These threads run in the background to perform tasks such as garbage collection. Daemon thread does allow JVM from existing until all the threads finish their execution. When a JVM founds daemon threads it terminates the thread and then shutdown itself, it does not care Daemon thread whether it is running or not.

**Thread Pool**  
In Java, is used for reusing the threads which were created previously for executing the current task. It ***also provides the solution if any problem occurs in the thread cycle or in resource thrashing***. In Java Thread pool a group of threads are created, one thread is selected and assigned job and after completion of job, it is sent back in the group.

**Thread Priorities**  
In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler. The priority is given by the JVM or by the programmer itself explicitly. The ***range of the priority is between 1 to 10*** and there are three variables which are static to define priority in a Thread Class.

**Note:**Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

# **Java Thread Class**

Thread class is the main class on which Java's Multithreading system is based. Thread class, along with its companion interface Runnable will be used to create and run threads for utilizing Multithreading feature of Java.  
It provides constructors and methods to support multithreading. It extends object class and implements Runnable interface.  
 **Signature of Thread class**

public class Thread extends Object implements Runnable**Thread Class Priority Constants**

|  |  |
| --- | --- |
| **Field** | **Description** |
| MAX\_PRIORITY | It represents the maximum priority that a thread can have. |
| MIN\_PRIORITY | It represents the minimum priority that a thread can have. |
| NORM\_PRIORITY | It represents the default priority that a thread can have. |

Constructors of Thread class

|  |  |
| --- | --- |
| Thread() | Thread(ThreadGroup group, Runnable target) |
| Thread(String str) | Thread(ThreadGroup group, Runnable target, String name) |
| Thread(Runnable r) | Thread(ThreadGroup group, Runnable target, String name, long stackSize) |
| Thread(Runnable r, String str) | Thread(ThreadGroup group, String name) |

**Thread Class Methods:** Thread class also defines many methods for managing threads. Some of them are,

|  |  |
| --- | --- |
| **Method** | **Description** |
| setName() | to give thread a name |
| getName() | return thread's name |
| getPriority() | return thread's priority |
| isAlive() | checks if thread is still running or not |
| join() | Wait for a thread to end |
| run() | Entry point for a thread |
| sleep() | suspend thread for a specified time |
| start() | start a thread by calling run() method |
| activeCount() | Returns an estimate of the number of active threads in the current thread's thread group and its subgroups. |
| checkAccess() | Determines if the currently running thread has permission to modify this thread. |
| currentThread() | Returns a reference to the currently executing thread object. |
| dumpStack() | Prints a stack trace of the current thread to the standard error stream. |
| getId() | Returns the identifier of this Thread. |
| getState() | Returns the state of this thread. |
| getThreadGroup() | Returns the thread group to which this thread belongs. |
| interrupt() | Interrupts this thread. |
| interrupted() | Tests whether the current thread has been interrupted. |
| isAlive() | Tests if this thread is alive. |
| isDaemon() | Tests if this thread is a daemon thread. |
| isInterrupted() | Tests whether this thread has been interrupted. |
| setDaemon(boolean on) | Marks this thread as either a daemon thread or a user thread. |
| setPriority(int newPriority) | Changes the priority of this thread. |
| yield() | A hint to the scheduler that the current thread is willing to yield its current use of a processor. |

**Some Important points to Remember**

1. When we extend Thread class, we cannot override **setName()** and **getName()** functions, because they are declared final in Thread class.
2. While using **sleep()**, always handle the exception it throws.

*static* void **sleep**(long *milliseconds*) throws **InterruptedException**

**Runnable Interface:** It also used to create thread and should be used if you are only planning to override the run() method and no other Thread methods.  
  
**Signature  
*@FunctionalInterface***public interface Runnable Interface **Method**

It provides only single method that must be implemented by the class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| run() | It runs the implemented thread. |

**Shutdown hook**In Java, Shutdown hook is used to clean-up all the resource, it means closing all the files, sending alerts etc. We can also save the state when the JVM shuts down. Shutdown hook mostly used when any code is to be executed before any JVM shuts down. Following are some of the reasons when the JVM shut down:

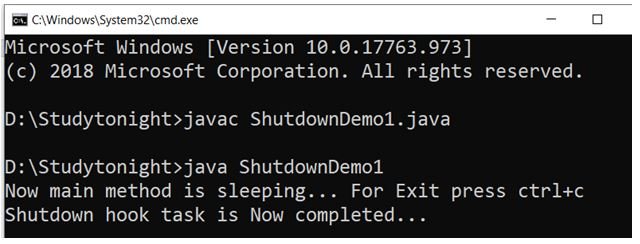
* Pressing ctrl+c on the command prompt
* When the System.exit(int) method is invoked.
* When user logoff or shutdown etc

#### **addShutdownHook(Thread hook)**

The addShutdownHook(Thread hook) method is used to register the thread with the virtual machine. This method is of Runtime class.

class Demo6 extends Thread{   
 public void run(){   
 System.out.println("Shutdown hook task is Now completed...");   
 }   
}

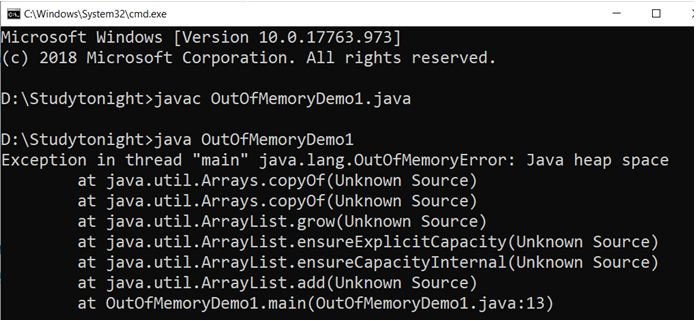
public class ShutdownDemo1{   
 public static void main(String[] args)throws Exception {   
 Runtime obj=Runtime.getRuntime();   
 obj.addShutdownHook(new Demo6());   
 System.out.println("Now main method is sleeping... For Exit press ctrl+c");   
 try{  
 Thread.sleep(4000);  
 }catch (Exception e) {}   
 }   
}

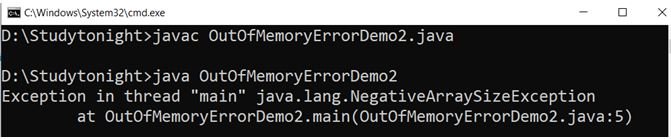


**OutOfMemory Exception:** In Java, as we know that all objects are stored in the heap. The objects are created using the new keyword. The OutOfMemoryError occurs as follow:

out-of-memory-exception

This error occurs when Java Virtual Machine is not able to allocate the object because it is out of memory and no memory can be available by the garbage collector.  
  
The meaning of OutOfMemoryError is that something wrong is in the program. Many times the problem can be out of control when the third party library caches strings.  
  
**Basic program in which OutOfMemoryError can occur  
Example:**import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
  
public class OutOfMemoryDemo1 {  
 public static void main(String[] args) {  
 List obj = new ArrayList<>();  
 Random obj1= new Random();  
 while (true)  
 obj.add(obj1.nextInt());  
 }  
}



Program in which OutOfMemoryError can occur because of low memory  
**Example:**public class OutOfMemoryErrorDemo2{  
 public static void main(String[] args) {  
 Integer[] a = new Integer[100000\*10000\*1000];  
 System.out.println("Done"); }}  


**StackOverflowError** happens when you execute too many methods one inside another (for example with an *infinite recursion*), which is limited by the size of the *stack*.  
 **OutOfMemoryError** happens when the JVM runs out of space to allocate new objects, which are allocated on the *heap*.  
  
**StackOverflowError**

* It is related to stack memory.
* It occurs when Stack is full.
* It is thrown when you call a method and there is no space left in the stack.
* It occurs when you are calling a method recursively without proper terminating condition.
* How to avoid? Make sure that methods are finishing their execution and leaving the stack memory.

**OutOfMemoryError**

* It is related to heap memory.
* It occurs when heap is full.
* It is thrown when you create a new object and there is no space left in the heap.
* It occurs when you are creating lots of objects in the heap memory.
* How to avoid? Try to remove references to objects which you don’t need anymore.

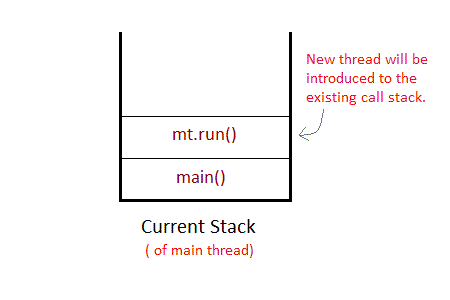
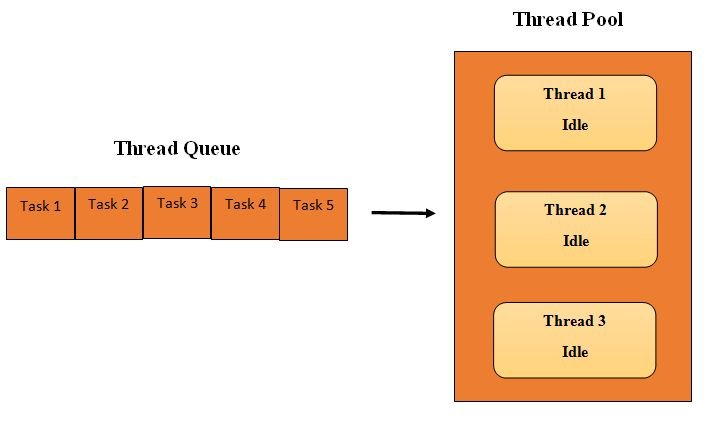
***Creating a thread in Java***

To implement multithreading, Java defines two ways by which a thread can be created.

* By implementing the **Runnable** interface.
* By extending the **Thread** class.

**What if we call run() method directly without using start() method?**In above program if we directly call run() method, without using start() method,

public static void main(String args[]){  
 MyThread mt = new MyThread();  
 mt.run();  
}

Doing so, the thread won't be allocated a new call stack, and it will start running in the current call stack, that is the call stack of the **main** thread. Hence Multithreading won't be there.  
  
  
  
**Can we start a thread twice?** **No**, a thread cannot be started twice. If you try to do so, ***IllegalThreadStateException*** will be thrown.  
  
**Thread Pool:** In Java, is used for reusing the threads which were created previously for executing the current task. It also provides the solution if any problem occurs in the thread cycle or in resource thrashing. In Java Thread pool a group of threads are created, one thread is selected and assigned job and after completion of job, it is sent back in the group.  


**There are four methods of a Thread pool. They are as following**:

1. newFixedThreadPool(int)  
2. newCachedThreadPool()  
3. newSingleThreadExecutor()  
4. newScheduledThreadPool()

**Following are the steps for creating a program of the thread pool**1. Create a runnable object to execute.  
2. Using executors create an executor pool  
3. Now pass the object to the executor pool  
4. At last shutdown the executor pool.  
  
Runtime.getRuntime().availableProcessors() 🡪 CPU Processor Count  
 **Example**:

import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
class WorkerThread implements Runnable{  
 private String message;  
  
 public WorkerThread(String a){  
 this.message=a;  
 }

public void run(){  
 System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);  
 processmessage();  
 System.out.println(Thread.currentThread().getName()+" (End)");  
 }

private void processmessage(){  
 try{   
 Thread.sleep(5000);   
 }catch (InterruptedException e){  
 System.out.println(e);   
} }

}

public class ThreadPoolDemo1{   
 public static void main(String[] args){  
 ExecutorService executor = Executors.newFixedThreadPool(5);  
 for (int i = 0; i < 10; i++){  
 Runnable obj = new WorkerThread("" + i);  
 executor.execute(obj);  
 }  
 executor.shutdown();  
 while (!executor.isTerminated()){  
 }  
 System.out.println("\*\*\*\*\*\*\*\*All threads are Finished\*\*\*\*\*\*\*\*");  
 }  
}

|  |  |  |
| --- | --- | --- |
| **Feature** | **execute(Runnable command)** | **submit(Callable<T> task) / submit(Runnable task)** |
| **Return Type** | void (Does not return a result) | Returns a Future<T> object |
| **Exception Handling** | Exceptions thrown in tasks will terminate the thread | Exceptions are captured inside Future.get() |
| **Usage** | Use when task **doesn’t need a result** | Use when task **returns a result** or you need **exception handling** |
| **Applicable Task Type** | Only Runnable | Both Runnable and Callable<T> |
| ***Blocking?*** (main Thread) | ***No*** *(Fire and forget)* | ***Yes****, when calling future.get()* |

**Summary**

* execute() → **Faster, fire-and-forget**, but **uncaught exceptions** terminate threads.
* submit() → **Returns a Future<T>**, allows **exception handling** and **task result retrieval**.

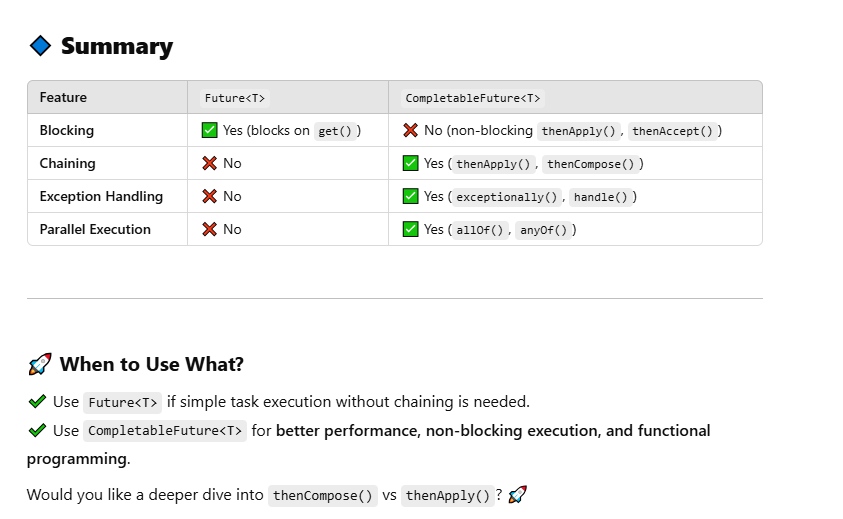
Submit or Execute When to Use Which?

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Use execute()** | **Use submit()** |
| Fire-and-forget execution | ✅ Yes | ❌ No |
| Need a return value | ❌ No | ✅ Yes |
| Handling exceptions in tasks | ❌ No | ✅ Yes |
| Runnable task with no return | ✅ Yes | ✅ Yes |
| Callable task with return value | ❌ No | ✅ Yes |

**Difference between Future and CompletableFuture in Java**

Both Future and CompletableFuture are used in asynchronous programming in Java. However, CompletableFuture provides more flexibility, better exception handling, and functional-style programming compared to Future.





# **Joining threads in Java**

Sometimes one thread needs to know when other thread is terminating. In java, **isAlive()** and **join()** are two different methods that are used to check whether a thread has finished its execution or not.  
  
The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

*final* boolean **isAlive()**

But, **join()** method is used more commonly than **isAlive()**. This method ***waits until the thread on which it is called terminates***.

*final* void **join()** throws **InterruptedException**

Using **join()** method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

*final* void **join**(long *milliseconds*) throws **InterruptedException**

As we have seen in the [Introduction to MultiThreading](http://www.studytonight.com/java/multithreading-in-java), the main thread must always be the last thread to finish its execution. Therefore, we can use Thread join() method to ensure that all the threads created by the program has been terminated before the execution of the main thread.

**Java Sleeping Thread**To sleep a thread for a specified time, Java provides sleep method which is defined in Thread class. The sleep method is an overloaded method which are given below. It throws interrupted exception so make sure to provide proper handler.

It always pause the current thread execution. Any other thread can interrupt the current thread in sleep, in that case InterruptedException is thrown.  
  
**Syntax**

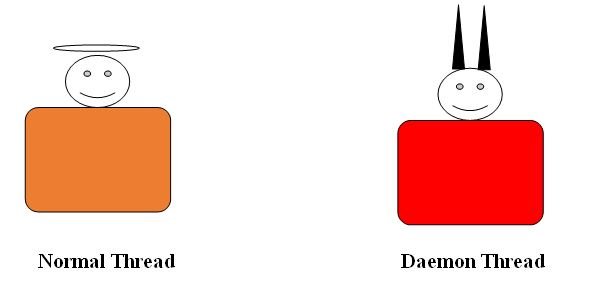
sleep(long millis)throws InterruptedException  
sleep(long millis, int nanos)throws InterruptedException  
 **Thread.sleep()** interacts with the thread scheduler to put the current thread in wait state for specified period of time. Once the wait time is over, thread state is changed to runnable state and wait for the CPU for further execution. So the actual time that current thread sleep depends on the thread scheduler that is part of operating system.  
 **Java Naming Thread**

Each thread in Java has its own name which is set by the JVM by default. Although there are many other attributes associated to the thread like: id, priority etc.

We can get name of a thread by calling getName() method of Thread class. If we wish to set new name of the thread then **setName()** method can be used. Both methods belong to Thread class and even we can set name of thread by passing into constructor during creating object.   
  
**Java Thread Priorities**

Priority of a thread describes how early it gets execution and selected by the thread scheduler. In Java, when we create a thread, always a priority is assigned to it. In a Multithreading environment, the processor assigns a priority to a thread scheduler. The priority is given by the JVM or by the programmer itself explicitly. The range of the priority is between 1 to 10 and there are three constant variables which are static and used to fetch priority of a Thread. They are as following:

1. public static int MIN\_PRIORITY It holds the minimum priority that can be given to a thread. The value for this is 1.  
2. public static int NORM\_PRIORITY It is the default priority that is given to a thread if it is not defined. The value for this is 0.  
3. public static int MAX\_PRIORITY It is the maximum priority that can be given to a thread. The value for this is 10.  
  
Get and Set methods in Thread priority  
1. public final intgetPriority()  
In Java, getPriority() method is in java.lang.Thread package. it is used to get the priority of a thread.  
  
2. public final void setPriority(intnewPriority)  
In Java setPriority(intnewPriority) method is in java.lang.Thread package. It is used to set the priority of a thread. The setPriority() method throws IllegalArgumentException if the value of new priority is above minimum and maximum limit.  
  
**Java Daemon Thread**  
Daemon threads is a low priority thread that provide supports to user threads. These threads can be user defined and system defined as well. Garbage collection thread is one of the system generated daemon thread that runs in background. These threads run in the background to perform tasks such as garbage collection. Daemon thread does allow JVM from existing until all the threads finish their execution. When a JVM founds daemon threads it terminates the thread and then shutdown itself, it does not care Daemon thread whether it is running or not.



**Following are the methods in Daemon Thread**

**1. void setDaemon(boolean status)**In Java, this method is used to create the current thread as a daemon thread or user thread. If there is a user thread as obj1 then obj1.setDaemon(true) will make it a Daemon thread and if there is a Daemon thread obj2 then calling obj2.setDaemon(false) will make it a user thread.  
  
**Syntax:**

public final void setDaemon(boolean on)

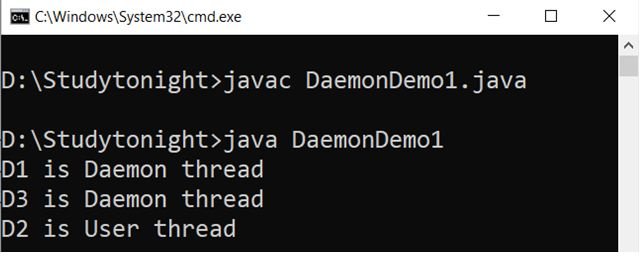
**2. boolean isDaemon()**In Java, this method is used to check whether the current thread is a daemon or not. It returns true if the thread is Daemon otherwise it returns false.  
**Syntax:** public final booleanisDaemon()

**Example:** Lets create an example to create daemon and user threads. To create daemon thread setdaemon() method is used. It takes boolean value either true or false.

public class DaemonDemo1 extends Thread {   
 public DaemonDemo1(String name1) {   
 super(name1);   
 }

public void run() {   
 if(Thread.currentThread().isDaemon()) {   
 System.out.println(getName() + " is Daemon thread");   
 } else {   
 System.out.println(getName() + " is User thread");   
 }   
 }

public static void main(String[] args) {   
 DaemonDemo1 D1 = new DaemonDemo1("D1");   
 DaemonDemo1 D2 = new DaemonDemo1("D2");   
 DaemonDemo1 D3 = new DaemonDemo1("D3");   
  
 D1.setDaemon(true);   
 D1.start();   
 D2.start();   
 D3.setDaemon(true);   
 D3.start();   
 }   
}



Example: **Daemon thread Priority**since daemon threads are low level threads then let’s check the priority of these threads. The priority we are getting is set by the JVM.

public class DaemonDemo1 extends Thread {   
 public DaemonDemo1(String name1) {   
 super(name1);   
 }

public void run() {   
 if(Thread.currentThread().isDaemon()) {   
 System.out.println(getName() + " is Daemon thread");   
 } else {   
 System.out.println(getName() + " is User thread");   
 }  
 System.out.println(getName()+" priority "+Thread.currentThread().getPriority());  
 }

public static void main(String[] args) {   
 DaemonDemo1 D1 = new DaemonDemo1("D1");   
 DaemonDemo1 D2 = new DaemonDemo1("D2");   
 DaemonDemo1 D3 = new DaemonDemo1("D3");

D1.setDaemon(true);   
 D1.start();   
 D2.start();   
 D3.setDaemon(true);   
 D3.start();   
 }   
}

D1 is Daemon thread  
D1 priority 5  
D2 is User thread  
D3 is Daemon thread  
D2 priority 5  
D3 priority 5  
  
**Example**While creating daemon thread make sure the setDaemon() is called before starting of the thread. Calling it after starting of thread will throw an exception and terminate the program execution.

public class DaemonDemo1 extends Thread {   
 public DaemonDemo1(String name1) {   
 super(name1);   
 }

public void run() {   
 if(Thread.currentThread().isDaemon()) {   
 System.out.println(getName() + " is Daemon thread");   
 } else {   
 System.out.println(getName() + " is User thread");   
 }  
 System.out.println(getName()+" priority "+Thread.currentThread().getPriority());  
 }

public static void main(String[] args) {   
 DaemonDemo1 D1 = new DaemonDemo1("D1");   
 DaemonDemo1 D2 = new DaemonDemo1("D2");   
 DaemonDemo1 D3 = new DaemonDemo1("D3");   
  
 D1.setDaemon(true);   
 D1.start();   
 D2.start();  
 D3.start();  
 D3.setDaemon(true);   
 }   
}

D1 is Daemon threadException in thread "main"   
D1 priority 5  
D3 is User thread  
D2 is User thread  
D2 priority 5java.lang.IllegalThreadStateException  
D3 priority 5  
 at java.base/java.lang.Thread.setDaemon(Thread.java:1410)  
 at myjavaproject.DaemonDemo1.main(DaemonDemo1.java:32)

**Java Synchronization**

Synchronization is a process of handling resource accessibility by multiple thread requests. The main purpose of synchronization is to avoid thread interference. At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. The synchronization keyword in java creates a block of code referred to as critical section.  
  
**Difference between synchronized keyword and synchronized block**When we use synchronized keyword with a method, it acquires a lock in the object for the whole method. It means that no other thread can use any synchronized method until the current thread, which has invoked it's synchronized method, has finished its execution.  
  
synchronized block acquires a lock in the object only between parentheses after the synchronized keyword. This means that no other thread can acquire a lock on the locked object until the synchronized block exits. But other threads can access the rest of the code of the method.

**Which is more preferred - Synchronized method or Synchronized block?**In Java, synchronized keyword causes a performance cost. A synchronized method in Java is very slow and can degrade performance. So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.  
 **Java Interthread Communication**Java provide benefits of avoiding thread pooling using inter-thread communication. The wait(), notify(), and notifyAll() methods of Object class are used for this purpose. These methods are implemented as **final** methods in Object, so that all classes have them. All the three method can be called only from within a **synchronized** context

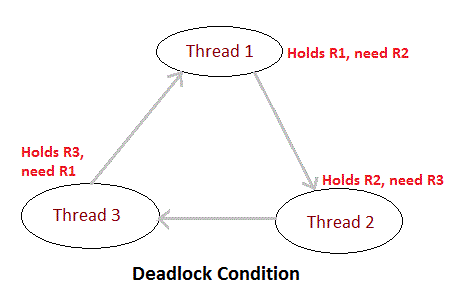
* wait() tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
* notify() wakes up a thread that called wait() on same object.
* notifyAll() wakes up all the thread that called wait() on same object.

Difference between wait() and sleep()

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| called from synchronized block | no such requirement |
| monitor is released | monitor is not released |
| gets awake when notify() or notifyAll() method is called. | does not get awake when notify() or notifyAll() method is called |
| not a static method | static method |
| wait() is generally used on condition | sleep() method is simply used to put your thread on sleep. |

Pooling is usually implemented by loop i.e. to check some condition repeatedly. Once condition is true appropriate action is taken. This waste CPU time.

**Thread Deadlock in Java**



Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.  
  
https://www.geeksforgeeks.org/different-ways-create-objects-java/  
 **Different ways to create objects in Java**  
  
1) **Using new Keyword**: Using new keyword is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way. By using this method we can call any constructor we want to call (no argument or parameterized constructors).  
  
2) **Using New Instance**: If we know the name of the class & if it has a public default constructor we can create an object –Class.forName. We can use it to create the Object of a Class. Class.forName actually loads the Class in Java but doesn’t create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.  
  
3) **Using clone() method**: Whenever clone() is called on any object, the JVM actually creates a new object and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.  
To use clone() method on an object we need to implement Cloneable and define the clone() method in it.  
  
4) **Using deserialization**: Whenever we serialize and then deserialize an object, JVM creates a separate object. In deserialization, JVM doesn’t use any constructor to create the object. To deserialize an object we need to implement the Serializable interface in the class.  
  
5) **Using newInstance() method of Constructor class** : This is similar to the newInstance() method of a class. There is one newInstance() method in the java.lang.reflect.Constructor class which we can use to create objects. It can also call parameterized constructor, and private constructor by using this newInstance() method.  
  
Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class internally uses newInstance() method of Constructor class.  
  
<https://codippa.com/how-to-break-singleton-in-java>   
  
**How to break Singleton class in java**  
1. ***Break by Cloning*** If a Singleton class implements java.lang.Cloneable interface then invoking clone() method on its single instance creates a duplicate object.  
  
**Remedy**: Don’t make the class which should be Singleton implement java.lang.Cloneable. If it extends a class which implements Cloneable, then override clone method and throw CloneNotSupportedException from it. This will prevent clone creation.  
  
2. ***Deserialization*** also breaks Singleton: Deserialization means bringing a saved object back to life. When a class is deserialized, a fresh instance of the class is created and its instance variables are then set to the values which were serialized.  
  
**Remedy**: Override readResolve method and return the same Singleton instance every time. readResolve method is called after the object has been read from the stream but before it is returned to the calling code.   
  
3. ***Reflection*** can instantiate a Singleton multiple times: Using java reflection api, we can tweak into a class by getting details like its fields, constructor, invoking its methods etc. Reflection can also be used to create new instance of a class. Obviously, if the class is Singleton, then creating a new instance again breaks its Singleton nature.  
  
**Remedy**: Check the static instance to be not null and throw Error if it is not null which means one instance already exists as .  
  
  
**Transaction Isolation Levels  
  
Transaction Isolation** defines how and when the changes made by one transaction become visible to other concurrent transactions. It’s a key concept in **ACID** properties — particularly **Isolation**. **Dirty Reads**: Transaction "A" writes a record. Meanwhile, Transaction "B" reads that same record before Transaction A commits. Later, Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ\_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.  
  
**Non-Repeatable Reads**: Transaction "A" reads some record. Then Transaction "B" writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.  
  
**Phantom Reads:**Transaction "A" reads a range of records. Meanwhile, Transaction "B" inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).  
  
**DEFAULT:**Use the default isolation level of the underlying database.  
**READ\_COMMITTED:**A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.  
**READ\_UNCOMMITTED:**This isolation level states that a transaction may read data that is still uncommitted by other transactions.  
**REPEATABLE\_READ:**A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.  
**SERIALIZABLE:**A constant indicating that dirty reads, non-repeatable reads, and phantom reads are prevented.  
  
<https://dzone.com/articles/spring-transaction-management>  
  
**Transaction Propagation**

|  |  |
| --- | --- |
| **Propagation** | **Behavior** |
| **REQUIRED** | **Always executes in a transaction.** If there is an existing transaction, it uses it. If none exists, then only a new one is created. |
| **SUPPORTS** | **It may or may not run in a transaction.** If the current transaction exists, then it is supported. If none exists, then it gets executed without a transaction. |
| **NOT\_SUPPORTED** | **Always executes without a transaction.** If there is an existing transaction, it gets suspended. |
| **REQUIRES\_NEW** | **Always executes in a new transaction.** If there is an existing transaction, it gets suspended. |
| **NEVER** | **Always executes without any transaction.** It throws an exception if there is an existing transaction |
| **MANDATORY** | **Always executes in a transaction.** If there is an existing transaction, it is used. If there is no existing transaction, it will throw an exception. |

**How HashSet works in Java**When you create an object of HashSet in Java, it internally creates an instance of backup Map with *default initial capacity 16* and *default load factor 0.75* as shown below :  
  
/\*\* Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75). \*/

public **HashSet**() {

map **=** **new** **HashMap**<>();

}

**How Object is stored in HashSet** As you can see below, a call to add(Object) is a delegate to put(Key, Value) internally, where Key is the object you have passed and value is another object,  called PRESENT, which is a constant in java.util.HashSet as shown below :  
 **private** transient **HashMap**<E, **Object**> map;

// Dummy value to associate with an Object in the backing Map

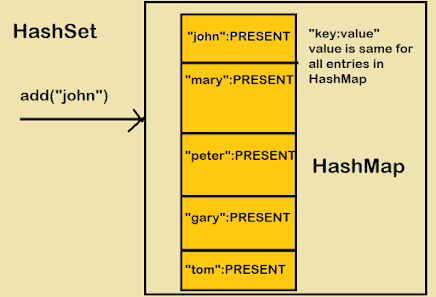
**private** static **final** **Object** **PRESENT** **=** **new** **Object**();

public boolean add(E e) {

**return** map.put(e, **PRESENT**)==**null**;

}

Since PRESENT is a constant, for all keys we have the same value in backup HashMap called map.

[](https://www.java67.com/2018/08/top-10-free-java-courses-for-beginners-experienced-developers.html)

**How Object is retrieved from HashSet** Now let's see the code for getting iterator for traversing over HashSet in Java. iterator() method from java.util.HashSet class returns iterator for backup Map **returned** by **map.keySet().iterator()** method.  
  
 /\*\*

\* Returns an iterator over the elements in this set. The elements are returned in no particular order.

\* @return an Iterator over the elements in this set @see ConcurrentModificationException

\*/

public **Iterator**<E> iterator() { **return** map.keySet().iterator(); }

**How to use HashSet in Java – Example**  
Using HashSet in Java is very simple, don't think it is Map but think more like Collection i.e. add elements by using add() method, check its return value to see if the object already existed in HashSet or not. Similarly use an iterator for retrieving elements from HashSet in Java.

You can also use contains() method to check if an object already exists in HashSet or not. This method use [equals() method](http://java67.blogspot.sg/2012/11/difference-between-operator-and-equals-method-in.html) for comparing object for matching. You can also use the remove() method to remove objects from HashSet. Since the element of HashSet is used as a key in backup HashMap, they must implement the [equals() and hashCode()](http://java67.blogspot.com/2013/04/example-of-overriding-equals-hashcode-compareTo-java-method.html) method.  
  
<https://www.java67.com/2016/01/difference-between-list-and-arraylist-variable-in-java.html#ixzz70loOwwDr>  
  
Why store ArrayList object on the List variable?

You might have seen something like this:  
**List**<Movie> listOfMovies = new **ArrayList**<Movie>()

The answer is to take advantage of Polymorphism. If you use interface than in the future if the new implementation is shipped, then you are not required to change your program.  
  
For example, an application written using **List** will work as expected whether you pass a [LinkedList](http://java67.blogspot.com/2016/01/how-to-implement-singly-linked-list-in-java-using-generics-example.html), [Vector](http://java67.blogspot.com/2012/09/arraylist-vs-vector-in-java-interview.html), or [ArrayList](http://java67.blogspot.com/2012/08/how-to-sort-arraylist-in-java-list.html) because they all implement List interface, they obey the contract exposed by the List interface  
  
  
**Difference between ArrayList and HashSet in Java**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Key** | **ArrayList** | **HashSet** |
| 1 | Implementation | ArrayList is the implementation of the list interface. | HashSet on the other hand is the implementation of a set interface. |
| 2 | Internal implementation | ArrayList internally implements Array for its implementation. | HashSet internally uses HashMap for its implementation. |
| 3 | Order of elements | ArrayList maintains the insertion order i.e., order of the object in which they are inserted. | HashSet is an unordered collection and doesn't maintain any order. |
| 4 | Duplicates | ArrayList allows duplicate values in its collection. | On other hand duplicate elements are not allowed in HashSet. |
| 5 | Index performance | ArrayList uses index for its performance i.e. its index based one can retrieve object by calling get(index) or remove objects by calling remove(index) | HashSet is completely based on object also it doesn't provide get() method. |
| 6 | Null Allowed | Any number of null value can be inserted in ArrayList without any restriction. | *HashSet allows only one null value in its collection*, after which no null value is allowed to be added. |

**Difference between HashTable, HashMap, ConcurrentHashMap**

|  |  |  |
| --- | --- | --- |
| **ConcurrentHashMap** | **SynchronizedMap** | **HashTable** |
| We will get thread safety without locking the total map object just with a bucket level lock. | We will get thread safety by locking the whole map object. | We will get thread safety by locking the whole map object |
| At a time multiple threads are allowed to operate on map objects safely. | At a time only one thread is allowed to perform any operation on a map object. | At a time one thread is allowed to operate on a map object. |
| Read operation can be performed without lock but write operation can be performed with bucket level lock. | Every read and write operations required total map object | Every read and write operations required total map object |
| While one thread iterating map objects the other thread is allowed to modify the map and won’t get ConcurrentModificationException. | While one thread iterating map object the other threads are not allowed to modify the map otherwise we will get ConcurrentModificationException | While one thread iterating map object the other threads are not allowed to modify the map otherwise we will get ConcurrentModificationException |
| Iterator of ConcurrentHashMap is fail-safe and won’t raise ConcurrentModificationException | Iterator of SynchronizedMap is fail-fast and it will raise ConcurrentModificationException | Iterator of HashTable is fail-fast and it will raise ConcurrentModificationException |
| ***Null is not allowed*** *for both keys and values.* | Null is allowed for both keys and values | Null is not allowed for both keys and values. |
| Introduce in java 1.5version | Introduce in java 1.2 version | Introduce in java 1.0version |

|  |
| --- |
| <https://beginnersbook.com/java-collections-tutorials>  enter image description here |
| Java Collections |

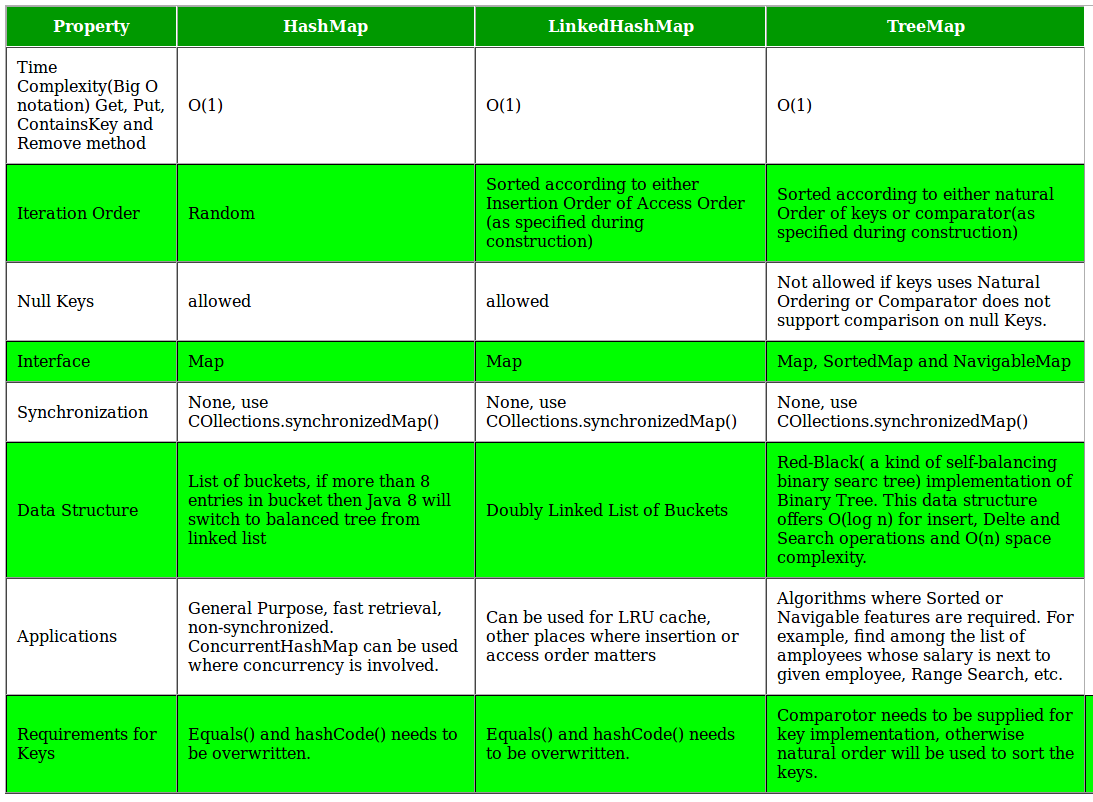
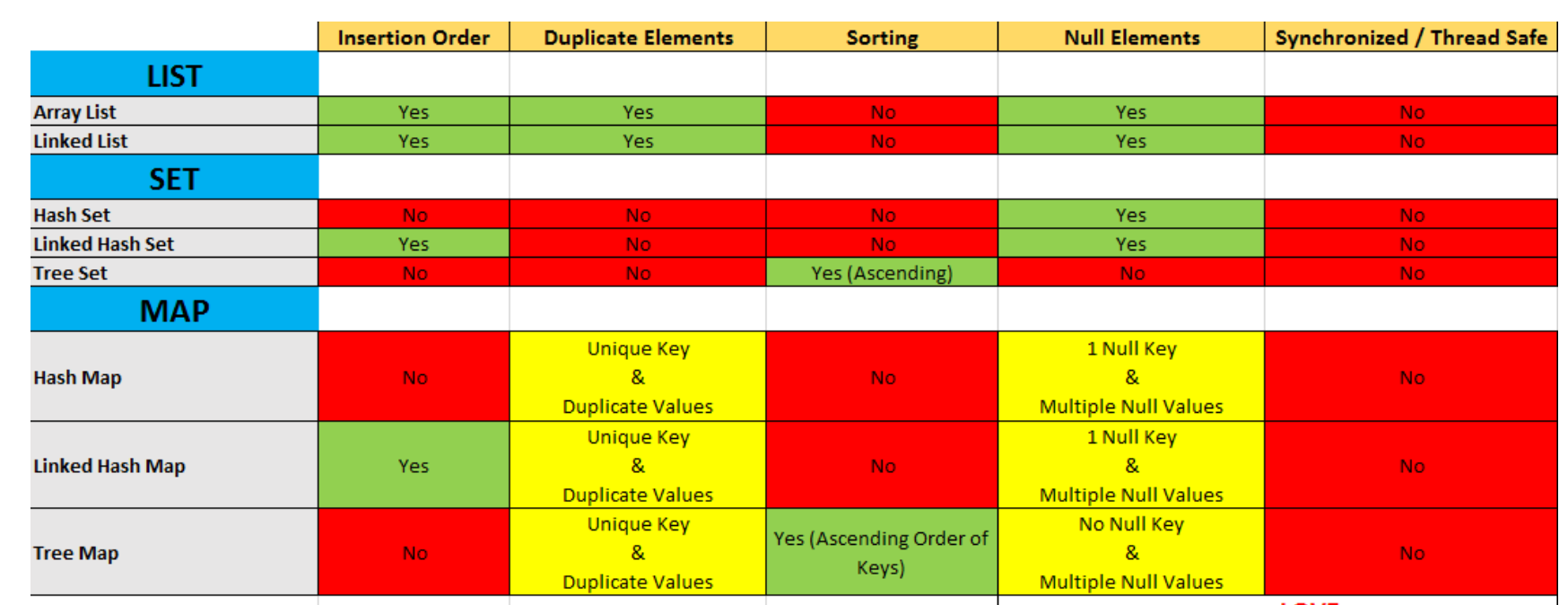
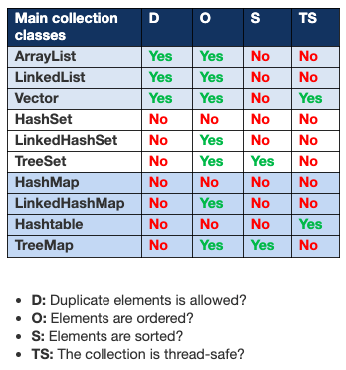
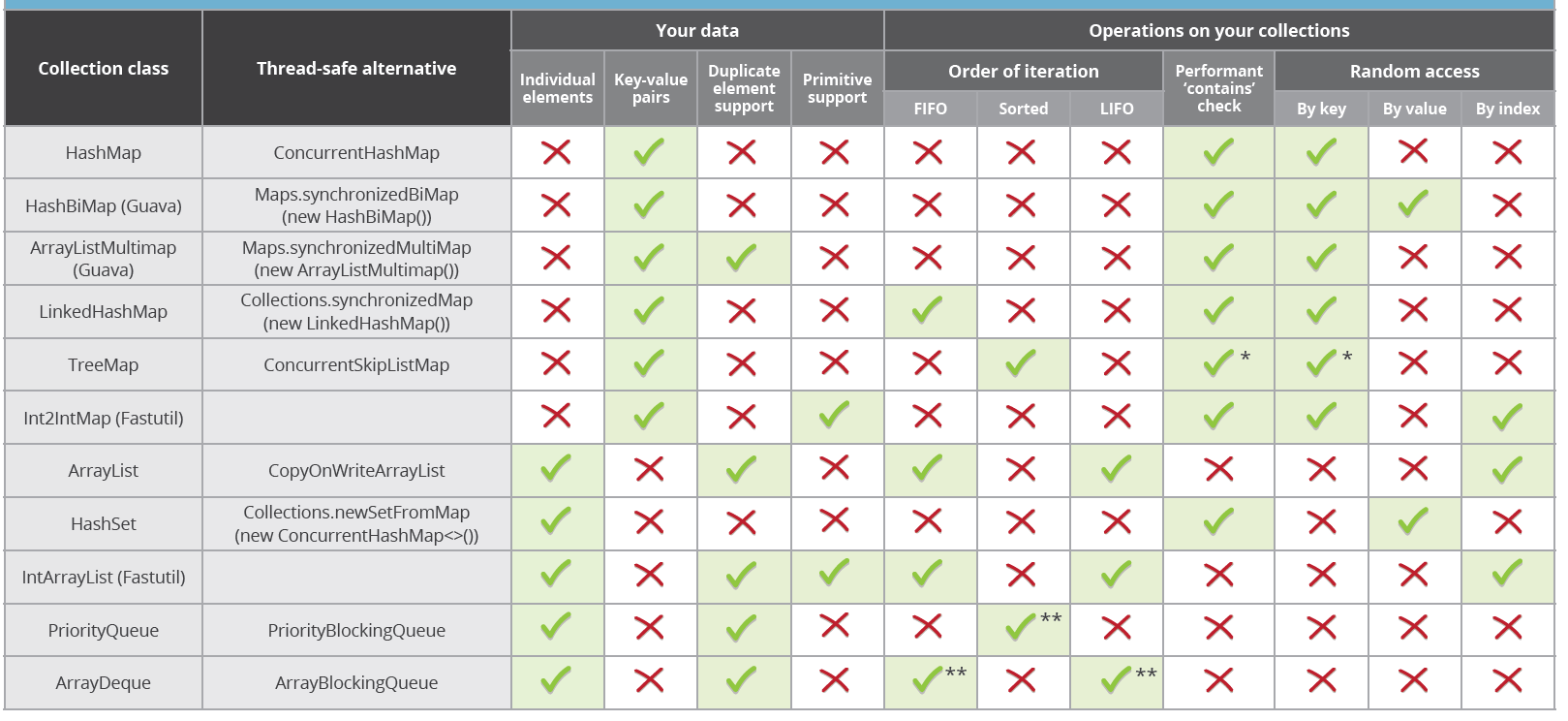
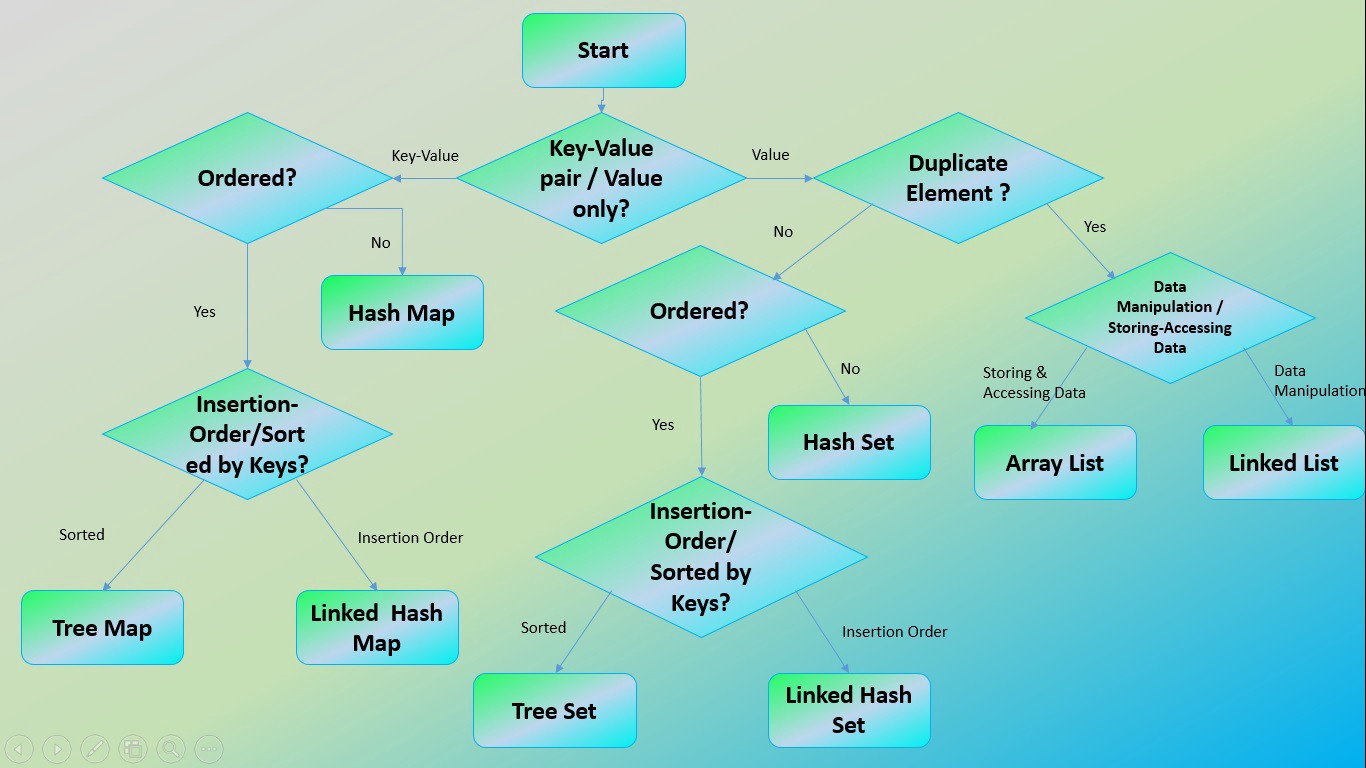
**Differences Between HashSet, LinkedHashSet and TreeSet In Java**

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **HashSet** | **LinkedHashSet** | **TreeSet** |
| How they work internally? | HashSet uses HashMap internally to store its elements. | LinkedHashSet uses LinkedHashMap internally to store its elements. | TreeSet uses TreeMap internally to store its elements. |
| Order Of Elements | HashSet doesn’t maintain any order of elements. | LinkedHashSet maintains insertion order of elements. I.e. elements are placed as they are inserted. | TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order. |
| Performance | HashSet gives better performance than the LinkedHashSet and TreeSet. | The performance of LinkedHashSet is between HashSet and TreeSet. Its performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements. | TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations. |
| Insertion, Removal And Retrieval Operations | HashSet gives performance of order O(1) for insertion, removal and retrieval operations. | LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations. | TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations. |
| How they compare the elements? | HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements. | LinkedHashSet also uses equals() and hashCode() methods to compare the elements. | TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements. It doesn’t use equals() and hashCode() methods for comparison of elements. |
| Null elements | HashSet allows maximum one null element. | LinkedHashSet also allows maximum one null element. | TreeSet doesn’t allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException. |
| Memory Occupation | HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements. | LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements. | TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap. |
| When To Use? | Use HashSet if you don’t want to maintain any order of elements. | Use LinkedHashSet if you want to maintain insertion order of elements. | Use TreeSet if you want to sort the elements according to some Comparator. |

**Similarities between HashSet, LinkedHashSet and TreeSet in Java:**

* All three doesn’t allow duplicate elements.
* All three are not synchronized.
* All three are Cloneable and Serializable.
* Iterator returned by all three is fail-fast in nature. I.e. you will get ConcurrentModificationException if they are modified after the creation of Iterator object.

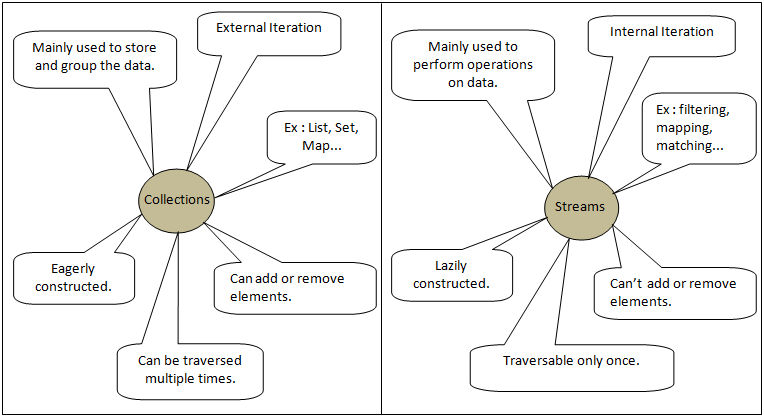
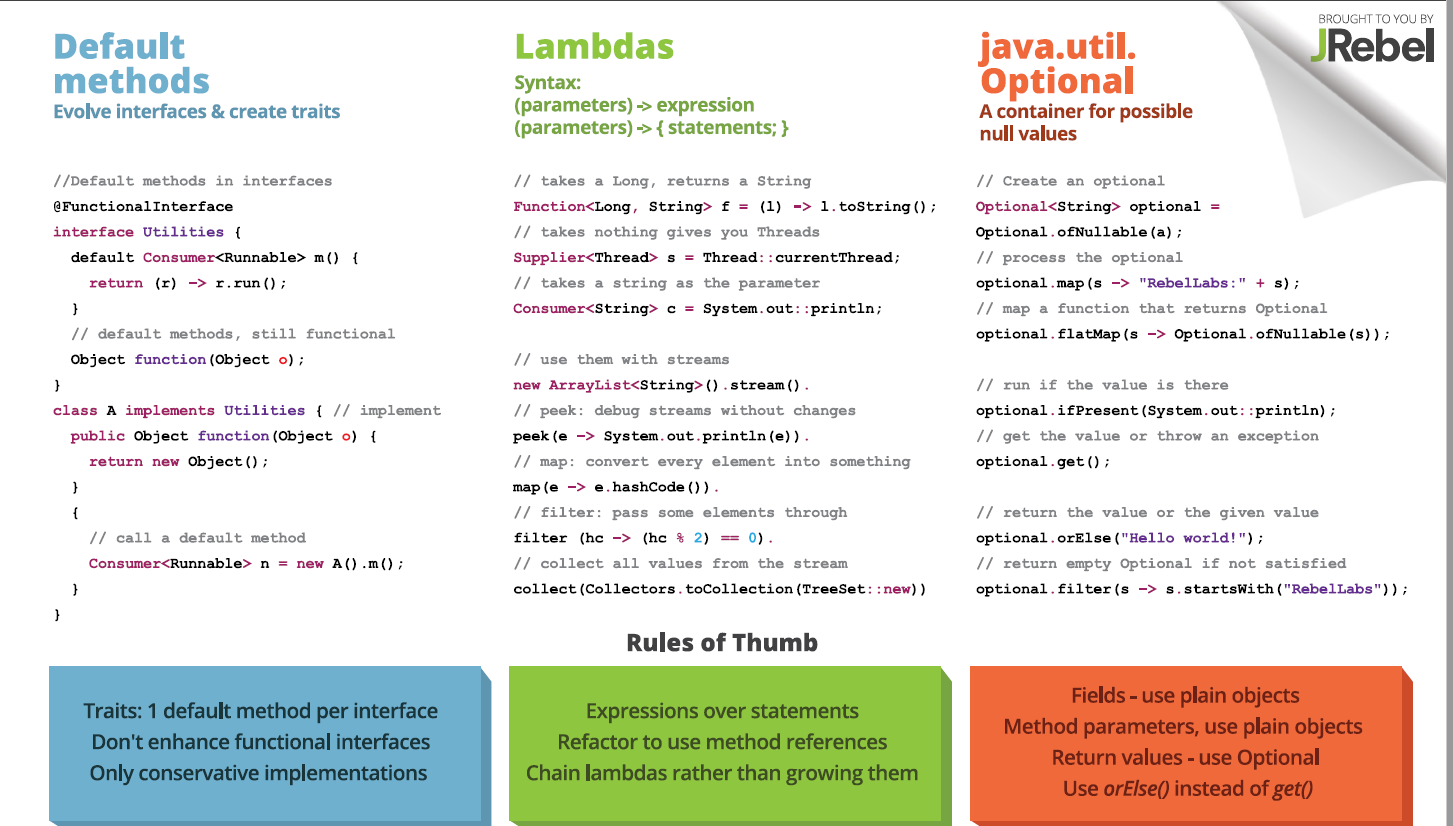
|  |  |
| --- | --- |
|  |  |

**Differences between TreeMap, HashMap and LinkedHashMap in Java**<https://www.geeksforgeeks.org/differences-treemap-hashmap-linkedhashmap-java/>   
  
  
  
  
  
  
  
  
  
  
  
  
  
**Difference Between Collections Vs Streams In Java**<https://javaconceptoftheday.com/collections-and-streams-in-java/>   
  
**1) Conceptual Difference**Collections are used to store and group the data in a particular data structure like **List**, **Set** or **Map**. But, streams are used to perform complex data processing operations like **filtering**, **matching**, **mapping** etc. on stored data such as arrays, collections or I/O resources. That means, collections are mainly about data and streams are mainly about operations on data.  
  
**2) Data Modification**  
You can add to or remove elements from collections. But, you can’t add to or remove elements from streams. Stream consumes a source, performs operations on it and returns a result. They don’t modify even the source also.  
  
**3) External Iteration Vs Internal Iteration**The main specialty of Java 8 Streams is that you need not to worry about iteration while using streams. Streams perform iteration internally behind the scene for you. You just have to mention the operations to be performed on a source.  
  
**4) Traversal**  
Streams are traversable only once. If you traverse the stream once, it is said to be consumed. To traverse it again, you have to get new stream from the source again. But, collections can be traversed multiple times.  
  
**5) Eager Construction Vs Lazy Construction**  
Collections are eagerly constructed i.e. all the elements are computed at the beginning itself. But, streams are lazily constructed i.e. intermediate operations are not evaluated until terminal operation is invoked.

|  |  |
| --- | --- |
| **Collections** | **Streams** |
| Collections are mainly used to store and group the data. | Streams are mainly used to perform operations on data. |
| You can add or remove elements from collections. | You can’t add or remove elements from streams. |
| Collections have to be iterated externally. | Streams are internally iterated. |
| Collections can be traversed multiple times. | Streams are traversable only once. |
| Collections are eagerly constructed. | Streams are lazily constructed. |
| Ex: List, Set, Map… | Ex: filtering, mapping, matching… |

**Difference between Collection and Collections: Collection vs Collections**

|  |  |  |
| --- | --- | --- |
|  | **Collection** | **Collections** |
| **Definition** | It is an interface that forms the root of the Java Collections Framework. | This utility class provides static methods to perform operations on collections. |
| **Usage** | Act as an interface for other interfaces like List, Set, and Queue. Some of the common methods include **add()**, **remove()**, and **size()**. | This class offers utility functions that operate on or return collections. For example, the **Collections.sort()** method sorts elements in a list. |
| **Implementation** | It doesn’t have any direct implementation as it is an interface. But you can implement the collection interface by using various Java classes, like ArrayList, HashSet, and PriorityQueue. | As it is a final class, meaning it cannot be sub classed. It doesn’t have any public constructors, so its methods are accessed statically. |
| **Flexibility** | Since it’s an interface, developers can create custom implementations if required. | As a utility class, it provides ready-to-use methods that help the developer to save time while performing common operations on collections. |
| **Purpose** | The purpose is to provide a standard way to handle and manage a group of objects. | It offers a utility function that simplifies common tasks associated with collections. |
| **package** | java.util | java.util.Collections |
| **Type** | ***Collection*** *is an Interface* | ***Collections*** *is a utility class* |

 **Jdk 1.8 Cheat Sheet**  
  
  
  
***Print values in a list in Uppercase***

Consumer<String> c = x->x.toUpperCase();

list.stream().foreach(y->c.accept(y));

https://www.youtube.com/watch?v=dQV6BeAlBvI&ab\_channel=CodeDecode

https://www.tutorialspoint.com/spring\_boot/spring\_boot\_interceptor.htm

https://www.youtube.com/watch?v=Hysb7hXp8B0&list=PLd3UqWTnYXOlHj\_fVVjUIVHjD5uEy2kk4&ab\_channel=DurgaSoftwareSolutions

<https://www.devglan.com/spring-boot/spring-boot-multiple-database-configuration>  
<https://docs.spring.io/spring-boot/docs/current/reference/html/actuator.html>  
frgconsulting.com/blog/java-interview-questions

https://www.educative.io/blog/getting-started-cors#what

**Spring JPA dynamic query example**

Simplest Explanation:

POST - Create NEW record

PUT - If the record exists update all the fields else, create a new record

PATCH - update the specific fields

GET - read

DELETE - delete

Why Would You Use POST Instead of GET for a Read Operation?  
<https://medium.com/swlh/why-would-you-use-post-instead-of-get-for-a-read-operation-381e4bdf3b9a>  
Even if it goes against the REST principles, if one doesn’t want to risk PII like phone numbers to be logged, then go ahead and use POST instead of GET.   
We can use POST instead of GET because sensitive information can be logged during the operation.  
  
<https://tcsglobal.udemy.com/course/java-8-new-features-in-simple-way/learn/lecture/8690398#overview>  
Java 8 New Features In Simple Way  
https://www.youtube.com/watch?v=BnknNTN8icw&ab\_channel=DailyCodeBuffer

https://www.youtube.com/watch?v=bspS-uTK0IM&ab\_channel=TechPrimers

https://www.youtube.com/playlist?list=PLqq-6Pq4lTTYTEooakHchTGglSvkZAjnE

https://www.youtube.com/watch?v=DduObGoRB9Y&ab\_channel=TechPrimers

https://www.javatpoint.com/log4j-example

https://www.infoworld.com/article/3373652/java-persistence-with-jpa-and-hibernate-part-1-entities-and-relationships.html

https://learnsql.com/blog/sql-join-interview-questions-with-answers/

https://snyk.io/blog/sum-list-of-numbers-in-java/

https://career.guru99.com/top-50-weblogic-interview-questions/

https://www.lambdatest.com/blog/top-29-microservices-interview-questions-for-2019/

https://www.knowledgehut.com/interview-questions/microservices

https://terasolunaorg.github.io/guideline/1.0.1.RELEASE/en/Overview/SpringMVCOverview.html#id1

/\*\*\*\*\*\*\*\*\*\*\*\*/

https://www.tutorialspoint.com/difference-between-bean-and-component-annotation-in-spring

https://www.softwaretestinghelp.com/design-patterns-in-java/

https://www.javatpoint.com/executor-framework-java

https://www.baeldung.com/circular-dependencies-in-spring

Spring Professional Certification Exam Tutorial - Module 01

https://www.baeldung.com/java-11-string-api

https://www.baeldung.com/java-11-new-features

https://www.youtube.com/watch?v=y04Niojm2NI&ab\_channel=ARJUNSINGHLODHI

https://www.naukri.com/job-listings-application-developer-microservices-ibm-india-pvt-limited-pune-8-to-10-years-210122906368?src=drecomm&sid=16431184512333433&xp=7&px=1

http://localhost:8080/restapi/webresources/myresource

https://www.javachinna.com/spring-aop-application-with-real-time-examples/

https://www.quora.com/Where-is-Spring-AOP-used-in-real-time

https://www.youtube.com/watch?v=DmFo5NioPME&ab\_channel=JavaTechies

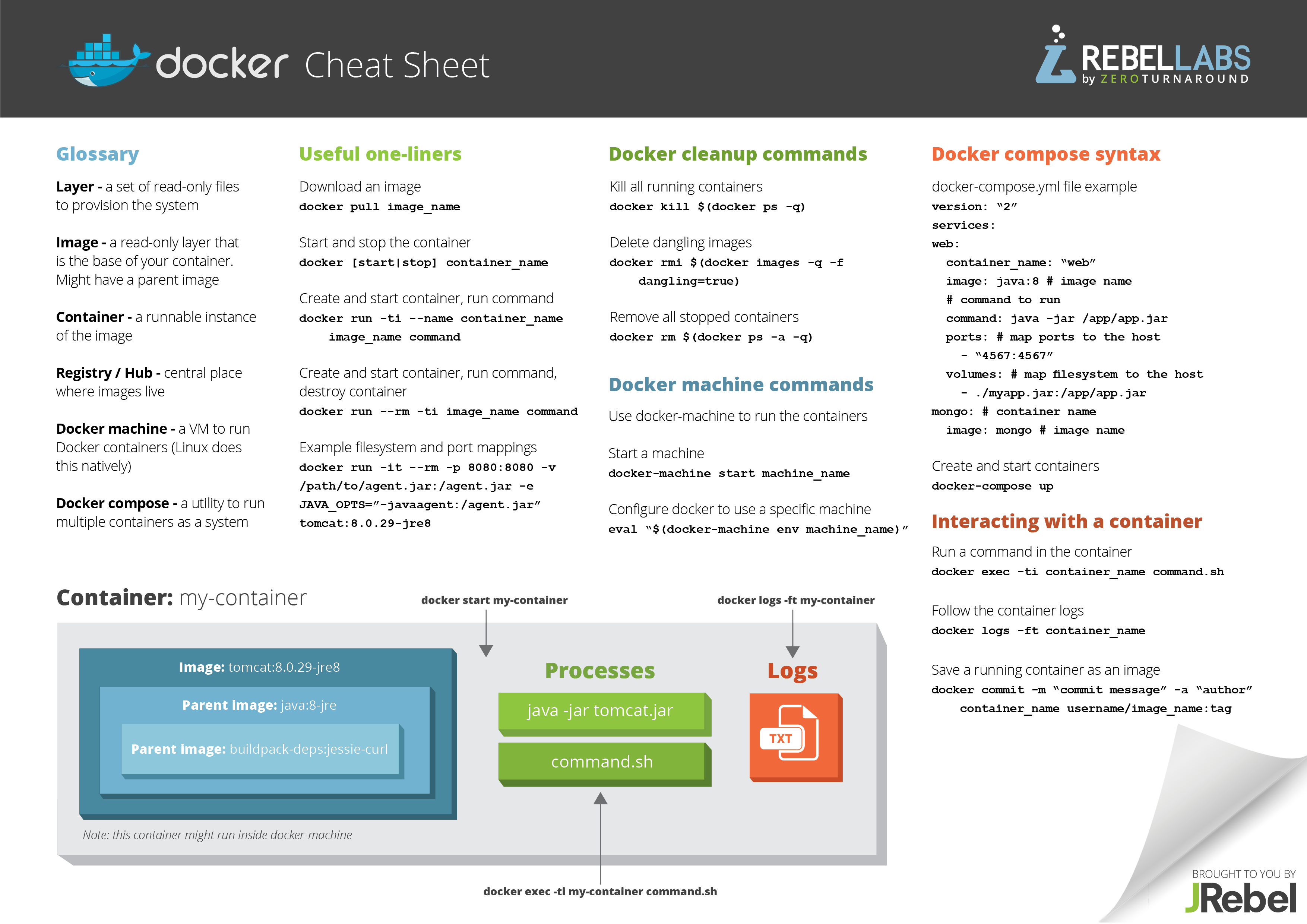
JPA CriteriaBuilder

https://www.netsurfingzone.com/jpa/jpa-criteriabuilder-example/

MongoDB Tutorial for Beginners 6 - Query Document

https://www.youtube.com/watch?v=FH4XRY6h8Fg&list=PLS1QulWo1RIZtR6bncmSaH8fB81oRl6MP&index=6&ab\_channel=ProgrammingKnowledge

https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

**Docker**:  


**SQL**:

|  |
| --- |
| sql, mysql, tutorials in a single image |
| Difference between LEFT and RIGHT OUTER Joins in SQL |

<https://www.pakainfo.com/mysql-full-outer-join/>

|  |  |
| --- | --- |
|  | mysql full outer join |
| www.complexsql.com/sql-joins-2/  SQL Join Example | https://www.complexsql.com/wp-content/uploads/2017/02/SQL-Inner-Join.jpg |

<https://javaconceptoftheday.com/java-8-stream-intermediate-and-terminal-operations/>  
 ***Java 8 Stream Intermediate and Terminal Operations***

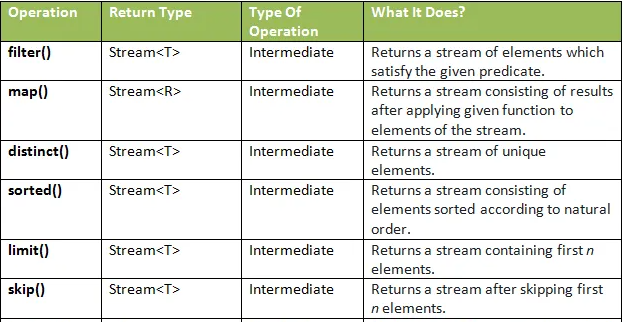
1) The main difference between intermediate and terminal operations is that intermediate operations return a stream as a result and terminal operations return non-stream values like primitive or object or collection or may not return anything.

2) As intermediate operations return another stream as a result, they can be chained together to form a pipeline of operations. Terminal operations cannot be chained together.

3) Pipeline of operations may contain any number of intermediate operations, but there has to be only one terminal operation, that too at the end of pipeline.

4) *Intermediate operations are lazily loaded. When you call intermediate operations, they are actually not executed. They are just stored in the memory and executed when the terminal operation is called on the stream.*

5) As the names suggest, intermediate operations doesn’t give end result. They just transform one stream to another stream. On the other hand, terminal operations give end result.

6)   
**Intermediate Operations:** map(), filter(), distinct(), sorted(), limit(), skip()  
**Terminal Operations:** forEach(), toArray(), reduce(), collect(), min(), max(),  count(), anyMatch(),  allMatch(), noneMatch(),   
findFirst(), findAny()  
  
list of intermediate and terminal operations.  
  
  
  
  
  
Stream.collect(Collectors.toList()), Stream.collect(Collectors.toUnmodifiableList()) and Stream.toList()

|  |  |  |
| --- | --- | --- |
| **Type** | **Guarantees unmodifiability** | **Allows Null** |
| Stream.collect(Collectors.toList()) | No | Yes |
| Stream.collect(Collectors.toUnmodifiableList()) | Yes | No |
| Stream.toList() | Yes | Yes |

***mvn clean package : Package command does below thing***

1. **Validate :-**Validate the project is correct and all necessary information is available or not .
2. **Compile :-**Compile your source code of the project.
3. **Test(optional) :-**Test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
4. **Package :-**Take the compiled code and package it as a jar or war as per pom file and move into target folder.
5. So when you run the command ***mvn package***, it runs the commands for all lifecycle phases till package.

mvn clean install: Install command does below thing:

1. **Validate :-**Validate the project is correct and all necessary information is available or not .
2. **Compile :-**Compile your source code of the project.
3. **Test(optional) :-**Test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed.
4. **Package :-**Take the compiled code and package it as a jar or war as per pom file and move into  target folder.
5. **verify** – Run any checks on results of integration tests to ensure quality criteria are met.
6. **install** – Install the package into the local repository, for use as a dependency in other projects locally.
7. **deploy** – Done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

So when you run command ***mvn install***, it runs the commands for all lifecycle phases till install, which includes package as well. So you can say, **install** phase comes after **package** phase.

* **mvn package** command will compile source code and also package it as a jar or war as per pom file and ***put it  into the target folder(by default).***
* **mvn install** command will compile and package, but it will also ***put the package in your local repository***. So that other projects can refer to it and grab it from your local repository.

|  |  |
| --- | --- |
|  | **mvn package** build, put it into **target folder**.  **mvn install** to create a JAR/WAR, put it into **local .m2**  **mvn deploy** to push artifacts to a **remote repository**. |

***SOLID Principles in Programming: Understand with Real Life Examples***  
<https://www.geeksforgeeks.org/solid-principle-in-programming-understand-with-real-life-examples/>

* Single Responsibility Principle (SRP) 🡪 ***A class should have only one reason to change***
* Open/Closed Principle 🡪 ***Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification***
* Liskov’s Substitution Principle (LSP) 🡪 ***Derived or child classes must be substitutable for their base or parent classes***
* Interface Segregation Principle (ISP) 🡪 ***Do not force any client to implement an interface which is irrelevant to them***
* Dependency Inversion Principle (DIP) 🡪 ***High-level modules should not depend on low-level modules. Both should depend on abstractions***

The SOLID principle helps in reducing tight coupling. Tight coupling means a group of classes are highly dependent on one another which you should avoid in your code.

* Opposite of tight coupling is loose coupling and your code is considered as a good code when it has loosely-coupled classes.
* Loosely coupled classes minimize changes in your code, helps in making code more reusable, maintainable, flexible and stable. Now let’s discuss one by one these principles…

## 1. Single Responsibility Principle

This principle states that “**A class should have only one reason to change**” which means every class should have a single responsibility or single job or single purpose. In other words, a class should have only one job or purpose within the software system.

Let’s understand Single Responsibility Principle using an example:

*Imagine a baker who is responsible for baking bread. The baker’s role is to focus on the task of baking bread, ensuring that the bread is of high quality, properly baked, and meets the bakery’s standards.*

* However, if the baker is also responsible for managing the inventory, ordering supplies, serving customers, and cleaning the bakery, this would violate the SRP.
* Each of these tasks represents a separate responsibility, and by combining them, the baker’s focus and effectiveness in baking bread could be compromised.
* To adhere to the SRP, the bakery could assign different roles to different individuals or teams. For example, there could be a separate person or team responsible for managing the inventory, another for ordering supplies, another for serving customers, and another for cleaning the bakery.

**2. Open/Closed Principle**

This principle states that “**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification**” which means you should be able to extend a class behavior, without modifying it.

Let’s understand Open/Closed Principle using an example:

*Imagine you have a class called PaymentProcessor that processes payments for an online store. Initially, the PaymentProcessor class only supports processing payments using credit cards. However, you want to extend its functionality to also support processing payments using PayPal.*

Instead of modifying the existing PaymentProcessor class to add PayPal support, you can create a new class called PayPalPaymentProcessor that extends the PaymentProcessor class. This way, the PaymentProcessor class remains closed for modification but open for extension, adhering to the Open-Closed Principle

**3. Liskov’s Substitution Principle**

The principle was introduced by Barbara Liskov in 1987 and according to this principle “**Derived or child classes must be substitutable for their base or parent classes**“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

Let’s understand Liskov’s Substitution Principle using an example:

*One of the classic examples of this principle is a rectangle having four sides. A rectangle’s height can be any value and width can be any value. A square is a rectangle with equal width and height. So we can say that we can extend the properties of the rectangle class into square class.*

In order to do that you need to swap the child (square) class with parent (rectangle) class to fit the definition of a square having four equal sides but a derived class does not affect the behavior of the parent class so if you will do that it will violate the Liskov’s Substitution Principle.

In simple words,

1. Objects of a superclass should be able to be replaced with objects of a subclass without affecting the program.
2. Object of subclass should be able to access the all the methods and properties of the superclass.

**Benefits of LSP:**

The benefit of following LSP is that it leads to more modular, extensible, and maintainable code. Here are some specific benefits:

1. **Code reusability**: When LSP is followed, the code can be reused easily because it allows for the use of generic interfaces or base classes that can be implemented or extended by different subclasses.
2. **Improved code quality**: By following LSP, the code becomes more understandable and easier to maintain.

**4. Interface Segregation Principle**

This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “**do not force any client to implement an interface which is irrelevant to them**“. Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

Let’s understand Interface Segregation Principle using an example:

*Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets.*

* In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don’t eat in your food. Here the menu should be different for different types of customers.
* The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

**5. Dependency Inversion Principle**

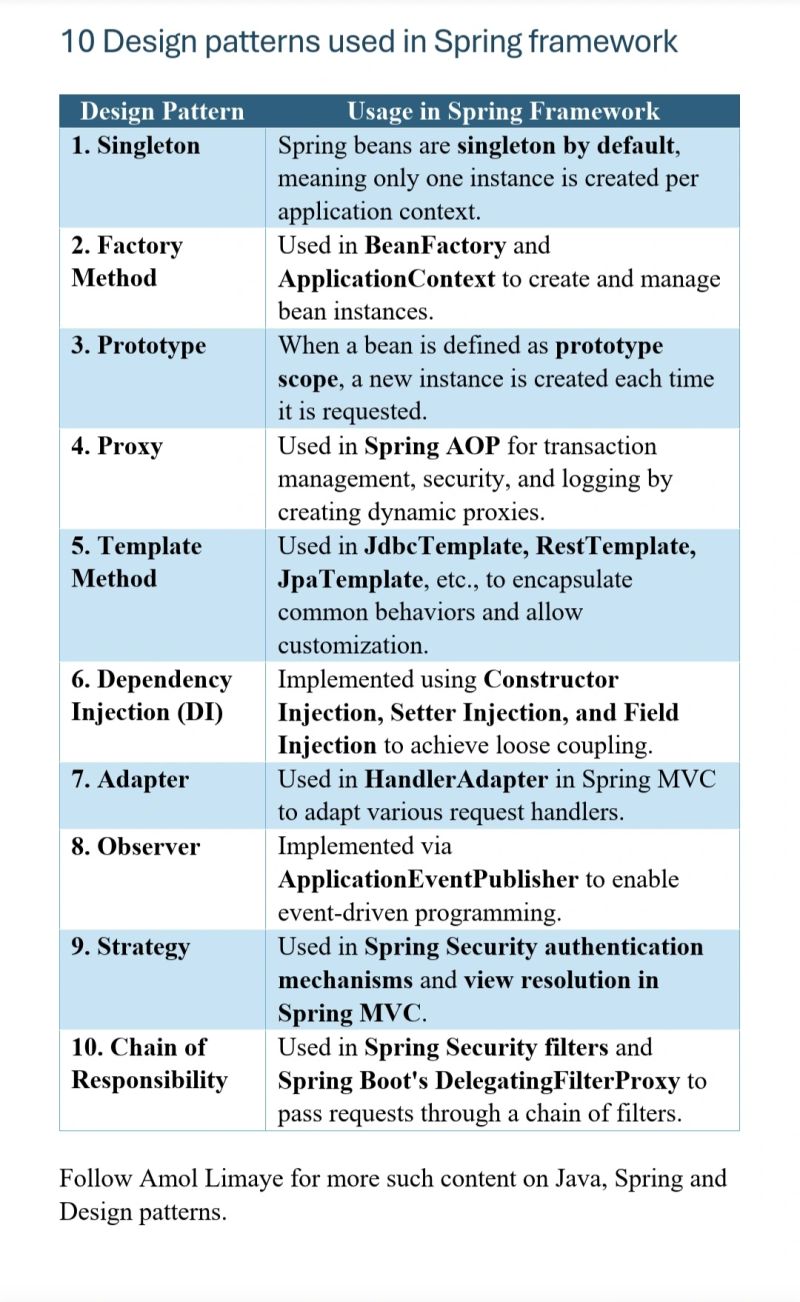
The Dependency Inversion Principle (DIP) is a principle in object-oriented design that states that “**High-level modules should not depend on low-level modules. Both should depend on abstractions**“. Additionally, abstractions should not depend on details. Details should depend on abstractions.

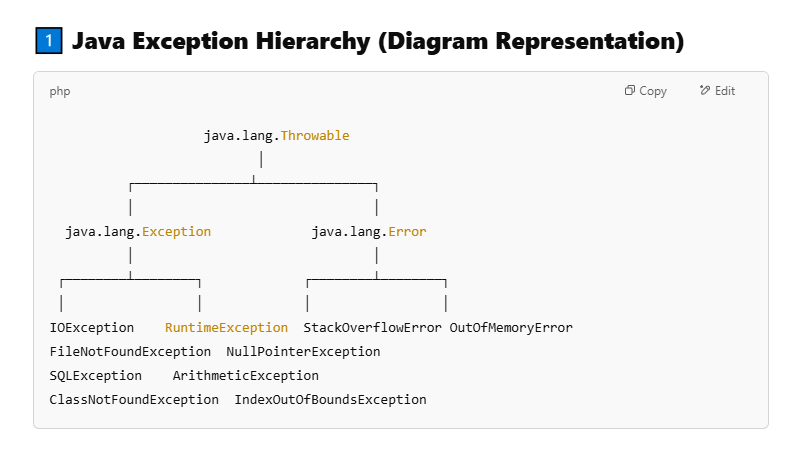
* In simpler terms, the DIP suggests that classes should rely on abstractions (e.g., interfaces or abstract classes) rather than concrete implementations.
* This allows for more flexible and decoupled code, making it easier to change implementations without affecting other parts of the codebase.

Let’s understand Dependency Inversion Principle using an example:

*In a software development team, developers depend on an abstract version control system (e.g., Git) to manage and track changes to the codebase. They don’t depend on specific details of how Git works internally.*

This allows developers to focus on writing code without needing to understand the intricacies of version control implementation.



***Exception Hierarchy*** 

***Exception (Recoverable Errors)***

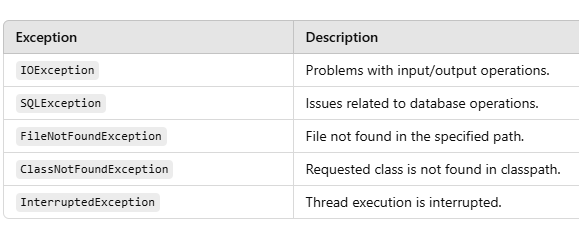
These are **checked or unchecked** exceptions that a program **should handle**.

**Checked Exceptions** → Compiler **forces** handling.

**Unchecked Exceptions** → Occur at runtime, usually due to programming mistakes.

***Checked Exceptions (Compile-Time Exceptions)***

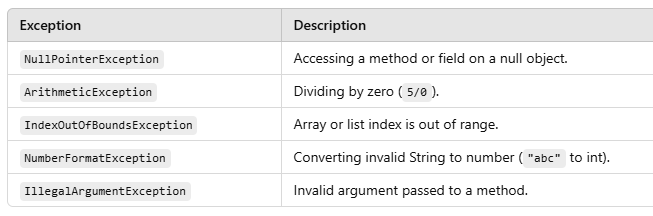
Must be handled using try-catch or declared using throws.

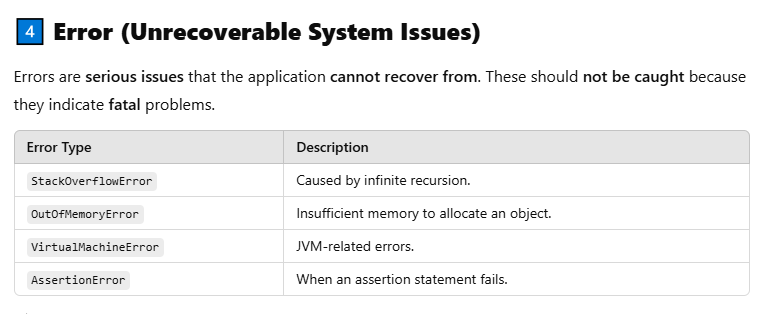
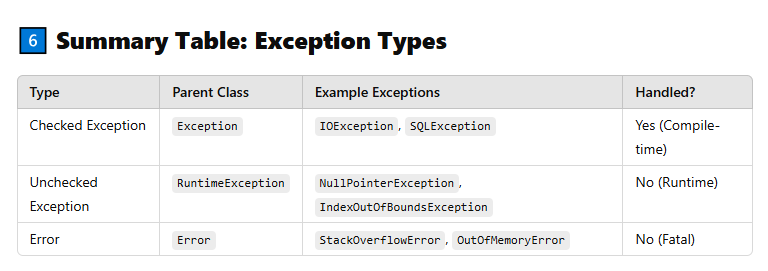
Occurs in scenarios where **external factors** cause failures.

***Unchecked Exceptions (Runtime Exceptions)***

Inherit from RuntimeException.

Not checked at compile-time but **occur due to programming mistakes**.



***Difference Between throw and throws in Java (Detailed Explanation)***

